do this, just use the Curve3D.sample_baked() method together with Curve2D.get_baked_length(). The first call to either of them will bake the curve internally.

Traversal at constant speed, then, can be done with the following pseudo-code:

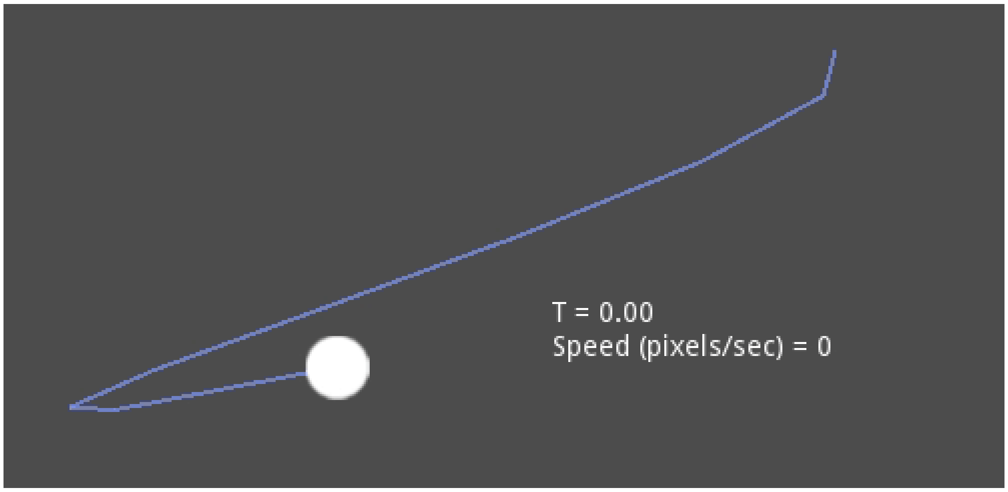
GDScript

```
var t = 0.0

func _process(delta):
    t += delta
    position = curve.sample_baked(t * curve.get_baked_length(),
true)
```

C#

```
private float _t = 0.0f;

public override void _Process(double delta)
{
    _t += (float)delta;
    Position = curve.SampleBaked(_t * curve.GetBakedLength(),
true);
}
```

And the output will, then, move at constant speed:



Random number generation

Many games rely on randomness to implement core game mechanics. This page guides you through common types of randomness and how to implement them in Godot.

After giving you a brief overview of useful functions that generate random numbers, you will learn how to get random elements from arrays, dictionaries, and how to use a noise generator in GDScript. Lastly, we'll take a look at cryptographically secure random number generation and how it differs from typical random number generation.

Note

Computers cannot generate "true" random numbers. Instead, they rely on pseudorandom number generators [https://en.wikipedia.org/wiki/Pseudorandom_number_generator] (PRNGs).

Godot internally uses the PCG Family [https://www.pcg-random.org/] of pseudorandom number generators.

Global scope versus RandomNumberGenerator class

Godot exposes two ways to generate random numbers: via *global scope* methods or using the RandomNumberGenerator class.

Global scope methods are easier to set up, but they don't offer as much control.

RandomNumberGenerator requires more code to use, but allows creating multiple instances, each with their own seed and state.

This tutorial uses global scope methods, except when the method only exists in the RandomNumberGenerator class.

The randomize() method

Note

Since Godot 4.0, the random seed is automatically set to a random value when the project starts. This means you don't need to call `randomize()` in `_ready()` anymore to ensure that results are random across project runs. However, you can still use `randomize()` if you want to use a specific seed number, or generate it using a different method.

In global scope, you can find a randomize() method. **This method should be called only once when your project starts to initialize the random seed.** Calling it multiple times is unnecessary and may impact performance negatively.

Putting it in your main scene script's `_ready()` method is a good choice:

GDScript

```
func _ready():
    randomize()
```

C#

```
public override void _Ready()
{
    GD.Randomize();
}
```

You can also set a fixed random seed instead using seed(). Doing so will give you *deterministic* results across runs:

GDScript

```
func _ready():
    seed(12345)
    # To use a string as a seed, you can hash it to a number.
    seed("Hello world".hash())
```

C#

```
public override void _Ready()
{
    GD.Seed(12345);
    GD.Seed("Hello world".Hash());
}
```

When using the RandomNumberGenerator class, you should call `randomize()` on the instance since it has its own seed:

GDScript

```
var random = RandomNumberGenerator.new()
random.randomize()
```

C#

```
var random = new RandomNumberGenerator();
random.Randomize();
```

Getting a random number

Let's look at some of the most commonly used functions and methods to generate random numbers in Godot.

The function randi() returns a random number between 0 and 2^32-1. Since the maximum value is huge, you most likely want to use the modulo operator (`%`) to bound the result between 0 and the denominator:

GDScript

```
# Prints a random integer between 0 and 49.
print(randi() % 50)

# Prints a random integer between 10 and 60.
print(randi() % 51 + 10)
```

C#

```
// Prints a random integer between 0 and 49.
GD.Print(GD.Randi() % 50);

// Prints a random integer between 10 and 60.
GD.Print(GD.Randi() % 51 + 10);
```

randf() returns a random floating-point number between 0 and 1. This is useful to implement a Weighted random probability system, among other things.

randfn() returns a random floating-point number following a normal distribution [https://en.wikipedia.org/wiki/Normal_distribution]. This means the returned value is more likely to be around the mean (0.0 by default), varying by the deviation (1.0 by default):

GDScript

```
# Prints a random floating-point number from a normal
distribution with a mean 0.0 and deviation 1.0.
var random = RandomNumberGenerator.new()
random.randomize()
print(random.randfn())
```

C#

```
// Prints a normally distributed floating-point number between
0.0 and 1.0.
var random = new RandomNumberGenerator();
random.Randomize();
GD.Print(random.Randfn());
```

randf_range() takes two arguments `from` and `to`, and returns a random floating-point number between `from` and `to`:

GDScript

```
# Prints a random floating-point number between -4 and 6.5.
print(randf_range(-4, 6.5))
```

RandomNumberGenerator.randi_range() takes two arguments `from` and `to`, and returns a random integer between `from` and `to`:

GDScript

```
# Prints a random integer between -10 and 10.
var random = RandomNumberGenerator.new()
random.randomize()
print(random.randi_range(-10, 10))
```

C#

```
// Prints a random integer number between -10 and 10.
random.Randomize();
GD.Print(random.RandiRange(-10, 10));
```

Get a random array element

We can use random integer generation to get a random element from an array:

GDScript

```
var _fruits = ["apple", "orange", "pear", "banana"]
```

```gdscript
func _ready():
    randomize()

    for i in range(100):
        # Pick 100 fruits randomly.
        print(get_fruit())


func get_fruit():
    var random_fruit = _fruits[randi() % _fruits.size()]
    # Returns "apple", "orange", "pear", or "banana" every time
the code runs.
    # We may get the same fruit multiple times in a row.
    return random_fruit
```

C#

```csharp
private string[] _fruits = { "apple", "orange", "pear",
"banana" };

public override void _Ready()
{
    GD.Randomize();

    for (int i = 0; i < 100; i++)
    {
        // Pick 100 fruits randomly.
        GD.Print(GetFruit());
    }
}

public string GetFruit()
{
    string randomFruit = _fruits[GD.Randi() % _fruits.Length];
    // Returns "apple", "orange", "pear", or "banana" every
time the code runs.
    // We may get the same fruit multiple times in a row.
    return randomFruit;
}
```

To prevent the same fruit from being picked more than once in a row, we can add more logic to this method:

GDScript

```
var _fruits = ["apple", "orange", "pear", "banana"]
var _last_fruit = ""


func _ready():
    randomize()

    # Pick 100 fruits randomly.
    for i in range(100):
        print(get_fruit())


func get_fruit():
    var random_fruit = _fruits[randi() % _fruits.size()]
    while random_fruit == _last_fruit:
        # The last fruit was picked, try again until we get a different fruit.
        random_fruit = _fruits[randi() % _fruits.size()]

    # Note: if the random element to pick is passed by reference,
    # such as an array or dictionary,
    # use `_last_fruit = random_fruit.duplicate()` instead.
    _last_fruit = random_fruit

    # Returns "apple", "orange", "pear", or "banana" every time the code runs.
    # The function will never return the same fruit more than once in a row.
    return random_fruit
```

C#

```
private string[] _fruits = { "apple", "orange", "pear", "banana" };
private string _lastFruit = "";
```

```
public override void _Ready()
{
    GD.Randomize();

    for (int i = 0; i < 100; i++)
    {
        // Pick 100 fruits randomly.
        GD.Print(GetFruit());
    }
}

public string GetFruit()
{
    string randomFruit = _fruits[GD.Randi() % _fruits.Length];
    while (randomFruit == _lastFruit)
    {
        // The last fruit was picked, try again until we get a different fruit.
        randomFruit = _fruits[GD.Randi() % _fruits.Length];
    }

    _lastFruit = randomFruit;

    // Returns "apple", "orange", "pear", or "banana" every time the code runs.
    // The function will never return the same fruit more than once in a row.
    return randomFruit;
}
```

This approach can be useful to make random number generation feel less repetitive. Still, it doesn't prevent results from "ping-ponging" between a limited set of values. To prevent this, use the shuffle bag pattern instead.

Get a random dictionary value

We can apply similar logic from arrays to dictionaries as well:

GDScript

```gdscript
var metals = {
    "copper": {"quantity": 50, "price": 50},
    "silver": {"quantity": 20, "price": 150},
    "gold": {"quantity": 3, "price": 500},
}


func _ready():
    randomize()

    for i in range(20):
        print(get_metal())


func get_metal():
    var random_metal = metals.values()[randi() % metals.size()]
    # Returns a random metal value dictionary every time the code runs.
    # The same metal may be selected multiple times in succession.
    return random_metal
```

Weighted random probability

The randf() method returns a floating-point number between 0.0 and 1.0. We can use this to create a "weighted" probability where different outcomes have different likelihoods:

GDScript

```gdscript
func _ready():
    randomize()

    for i in range(100):
        print(get_item_rarity())


func get_item_rarity():
    var random_float = randf()

    if random_float < 0.8:
        # 80% chance of being returned.
        return "Common"
    elif random_float < 0.95:
        # 15% chance of being returned.
        return "Uncommon"
    else:
        # 5% chance of being returned.
        return "Rare"
```

C#

```
public override void _Ready()
{
    GD.Randomize();

    for (int i = 0; i < 100; i++)
    {
        GD.Print(GetItemRarity());
    }
}

public string GetItemRarity()
{
    float randomFloat = GD.Randf();

    if (randomFloat < 0.8f)
    {
        // 80% chance of being returned.
        return "Common";
    }
    else if (randomFloat < 0.95f)
    {
        // 15% chance of being returned
        return "Uncommon";
    }
    else
    {
        // 5% chance of being returned.
        return "Rare";
    }
}
```

"Better" randomness using shuffle bags

Taking the same example as above, we would like to pick fruits at random. However, relying on random number generation every time a fruit is selected can lead to a less *uniform* distribution. If the player is lucky (or unlucky), they could get the same fruit three or more times in a row.

You can accomplish this using the *shuffle bag* pattern. It works by removing an element from the array after choosing it. After multiple selections, the array ends up empty. When that happens, you reinitialize it to its default value:

```
var _fruits = ["apple", "orange", "pear", "banana"]
# A copy of the fruits array so we can restore the original
value into `fruits`.
var _fruits_full = []


func _ready():
    randomize()
    _fruits_full = _fruits.duplicate()
    _fruits.shuffle()

    for i in 100:
        print(get_fruit())


func get_fruit():
    if _fruits.is_empty():
        # Fill the fruits array again and shuffle it.
        _fruits = _fruits_full.duplicate()
        _fruits.shuffle()
    # Get a random fruit, since we shuffled the array,
    # and remove it from the `_fruits` array.
    var random_fruit = _fruits.pop_front()
    # Prints "apple", "orange", "pear", or "banana" every time
the code runs.
    return random_fruit
```

When running the above code, there is a chance to get the same fruit twice in a row. Once we picked a fruit, it will no longer be a possible return value unless the array is now empty. When the array is empty, we reset it back to its default value, making it possible to have the same fruit again, but only once.

Random noise

The random number generation shown above can show its limits when you need a value that *slowly* changes depending on the input. The input can be a position, time, or anything else.

To achieve this, you can use random *noise* functions. Noise functions are especially popular in procedural generation to generate realistic-looking terrain. Godot provides FastNoiseLite for this, which supports 1D, 2D and 3D noise. Here's an example with 1D noise:

GDScript

```
var _noise = FastNoiseLite.new()

func _ready():
    randomize()
    # Configure the FastNoiseLite instance.
    _noise.noise_type =
FastNoiseLite.NoiseType.TYPE_SIMPLEX_SMOOTH
    _noise.seed = randi()
    _noise.fractal_octaves = 4
    _noise.frequency = 1.0 / 20.0
    for i in 100:
        # Prints a slowly-changing series of floating-point
numbers
        # between -1.0 and 1.0.
        print(_noise.get_noise_1d(i))
```

C#

```
private FastNoiseLite _noise = new FastNoiseLite();

public override void _Ready()
{
    GD.Randomize();
    // Configure the FastNoiseLite instance.
    _noise.NoiseType = NoiseTypeEnum.SimplexSmooth;
    _noise.Seed = (int)GD.Randi();
    _noise.FractalOctaves = 4;
    _noise.Frequency = 1.0f / 20.0f;

    for (int i = 0; i < 100; i++)
    {
        GD.Print(_noise.GetNoise1D(i));
    }
}
```

Cryptographically secure pseudorandom number generation

So far, the approaches mentioned above are **not** suitable for *cryptographically secure* pseudorandom number generation (CSPRNG). This is fine for games, but this is not sufficient for scenarios where encryption, authentication or signing is involved.

Godot offers a Crypto class for this. This class can perform asymmetric key encryption/decryption, signing/verification, while also generating cryptographically secure random bytes, RSA keys, HMAC digests, and self-signed X509Certificates.

The downside of CSPRNG is that it's much slower than standard pseudorandom number generation. Its API is also less convenient to use. As a result, CSPRNG should be avoided for gameplay elements.

Example of using the Crypto class to generate 2 random integers between `0` and `2^32 - 1` (inclusive):

```
var crypto := Crypto.new()
# Request as many bytes as you need, but try to minimize the
amount
# of separate requests to improve performance.
# Each 32-bit integer requires 4 bytes, so we request 8 bytes.
var byte_array := crypto.generate_random_bytes(8)

# Use the ``decode_u32()`` method from PackedByteArray to
decode a 32-bit unsigned integer
# from the beginning of `byte_array`. This method doesn't
modify `byte_array`.
var random_int_1 := byte_array.decode_u32(0)
# Do the same as above, but with an offset of 4 bytes since
we've already decoded
# the first 4 bytes previously.
var random_int_2 := byte_array.decode_u32(4)

prints("Random integers:", random_int_1, random_int_2)
```

See also

See PackedByteArray's documentation for other methods you can use to decode the generated bytes into various types of data, such as integers or floats.

Navigation

- 2D navigation overview
- 3D navigation overview
- Using NavigationServer
- Using NavigationMaps
- Using NavigationRegions
- Using navigation meshes
- Using NavigationPaths
- Using NavigationPathQueryObjects
- Using NavigationAgents
- Using NavigationObstacles
- Using NavigationLinks
- Using NavigationLayers
- Navigation debug tools
- Connecting navigation meshes
- Support different actor types
- Support different actor locomotion
- Support different actor area access
- Optimizing Navigation Performance

2D navigation overview

Godot provides multiple objects, classes and servers to facilitate grid-based or mesh-based navigation and pathfinding for 2D and 3D games. The following section provides a quick overview over all available navigation related objects in Godot for 2D scenes and their primary use.

Godot provides the following objects and classes for 2D navigation:

- Astar2D

  `Astar2D` objects provide an option to find the shortest path in a graph of weighted **points**.

  The AStar2D class is best suited for cell-based 2D gameplay that does not require actors to reach any possible position within an area but only predefined, distinct positions.

- NavigationServer2D

  `NavigationServer2D` provides a powerful server API to find the shortest path between two positions on an area defined by a navigation mesh.

  The NavigationServer is best suited for 2D realtime gameplay that does require actors to reach any possible position within a navigation mesh defined area. Mesh-based navigation scales well with large game worlds as a large area can often be defined with a single polygon when it would require many, many grid cells.

  The NavigationServer holds different navigation maps that each consist of regions that hold navigation mesh data. Agents can be placed on a map for avoidance calculation.

RIDs are used to reference internal maps, regions, and agents when communicating with the server.

The following NavigationServer RID types are available.

  - NavMap RID
    Reference to a specific navigation map that holds regions and agents. The map will attempt to join the navigation meshes of the regions by proximity. The map will synchronize regions and agents each physics frame.
  - NavRegion RID
    Reference to a specific navigation region that can hold navigation mesh data. The region can be enabled / disabled or the use restricted with a navigation layer bitmask.
  - NavLink RID
    Reference to a specific navigation link that connects two navigation mesh positions over arbitrary distances.
  - NavAgent RID
    Reference to a specific avoidance agent. The avoidance is specified by a radius value.
  - NavObstacle RID
    Reference to a specific avoidance obstacle used to affect and constrain the avoidance velocity of agents.

The following scene tree nodes are available as helpers to work with the NavigationServer2D API.

- NavigationRegion2D Node
  A Node that holds a NavigationPolygon resource that defines a navigation mesh for the NavigationServer2D.

    - The region can be enabled / disabled.
    - The use in pathfinding can be further restricted through the `navigation_layers` bitmask.
    - The NavigationServer2D will join the navigation meshes of regions by proximity for a combined navigation mesh.
- NavigationLink2D Node

  A Node that connects two positions on navigation meshes over arbitrary distances for pathfinding.

    - The link can be enabled / disabled.
    - The link can be made one-way or bidirectional.
    - The use in pathfinding can be further restricted through the `navigation_layers` bitmask.

  Links tell the pathfinding that a connection exists and at what cost. The actual agent handling and movement needs to happen in custom scripts.

- NavigationAgent2D Node

  A helper Node used to facilitate common NavigationServer2D API calls for pathfinding and avoidance. Use this Node with a Node2D inheriting parent Node.

- NavigationObstacle2D Node

  A Node that can be used to affect and constrain the avoidance velocity of avoidance enabled agents. This Node does NOT affect the pathfinding of agents. You need to change the navigation meshes for that instead.

The 2D navigation meshes are defined with the following resources:

- NavigationPolygon Resource

  A resource that holds 2D navigation mesh data. It provides polygon drawing tools to allow defining navigation areas inside the Editor as well as at runtime.
    - The NavigationRegion2D Node uses this resource to define its navigation area.

  - The NavigationServer2D uses this resource to update the navigation mesh of individual regions.
  - The TileSet Editor creates and uses this resource internally when defining tile navigation areas.

See also

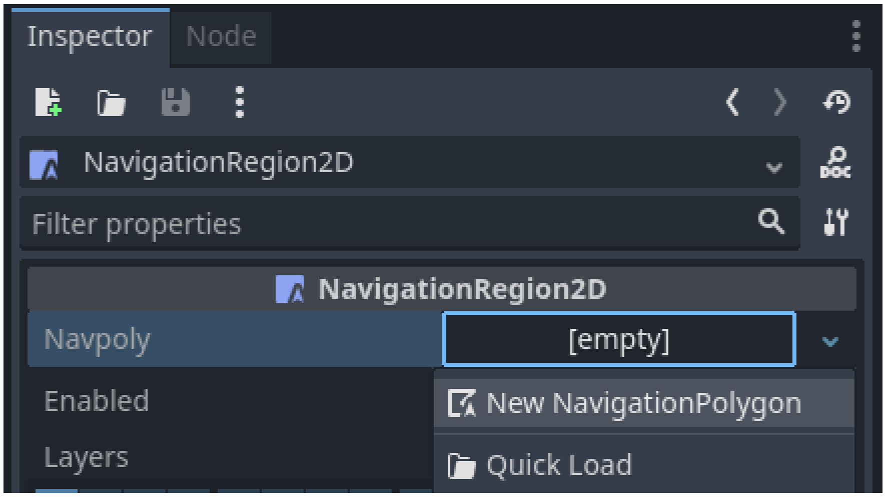
You can see how 2D navigation works in action using the 2D Navigation Polygon [https://github.com/godotengine/godot-demo-projects/tree/master/2d/navigation] and Grid-based Navigation with AStarGrid2D [https://github.com/godotengine/godot-demo-projects/tree/master/2d/navigation_astar] demo projects.

Setup for 2D scene

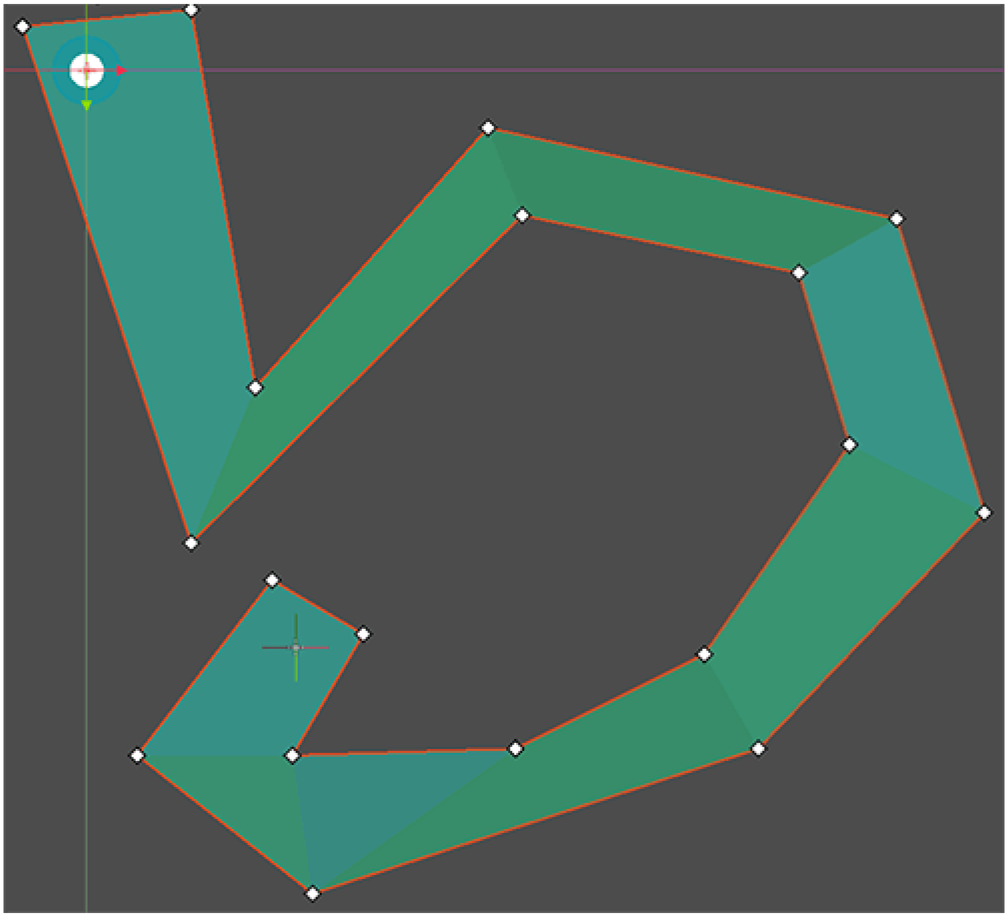
The following steps show the basic setup for minimal viable navigation in 2D. It uses the NavigationServer2D and a NavigationAgent2D for path movement.

1. Add a NavigationRegion2D Node to the scene.

2. Click on the region node and add a new NavigationPolygon Resource to the region node.


Inspector
Node
NavigationRegion2D
Filter properties
NavigationRegion2D
Navpoly
[empty]
Enabled
Layers
New NavigationPolygon
Quick Load

3. Define the movable navigation area with the NavigationPolygon draw tool. Then click the *Bake NavigationPolygon*` button on the toolbar.

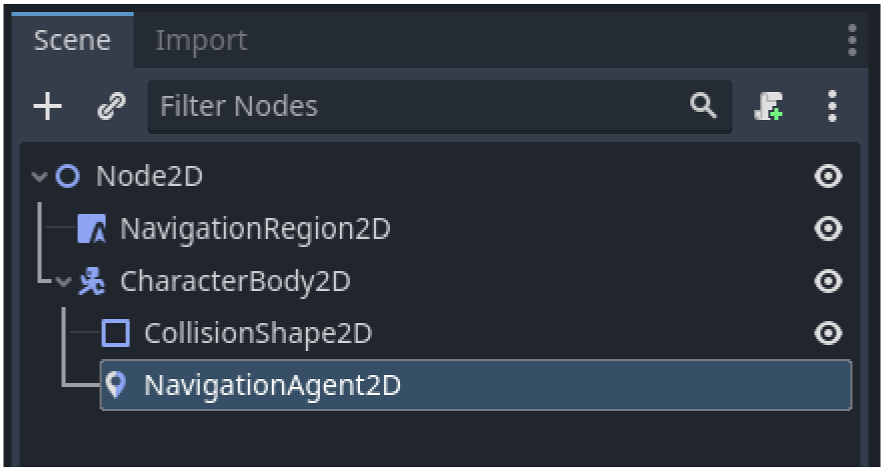


Note

The navigation mesh defines the area where an actor can stand and move with its center. Leave enough margin between the navigation polygon edges and collision objects to not get path following actors repeatedly stuck on collision.

4. Add a CharacterBody2D node in the scene with a basic collision shape and a sprite or mesh for visuals.

5. Add a NavigationAgent2D node below the character node.


Scene
Import
Filter Nodes
Node2D
NavigationRegion2D
CharacterBody2D
CollisionShape2D
NavigationAgent2D

6. Add the following script to the CharacterBody2D node. We make sure to set a movement target after the scene has fully loaded and the NavigationServer had time to sync.

GDScript

```
extends CharacterBody2D

var movement_speed: float = 200.0
var movement_target_position: Vector2 = Vector2(60.0,180.0)

@onready var navigation_agent: NavigationAgent2D =
$NavigationAgent2D

func _ready():
    # These values need to be adjusted for the actor's speed
    # and the navigation layout.
    navigation_agent.path_desired_distance = 4.0
    navigation_agent.target_desired_distance = 4.0

    # Make sure to not await during _ready.
    actor_setup.call_deferred()

func actor_setup():
    # Wait for the first physics frame so the NavigationServer
can sync.
    await get_tree().physics_frame

    # Now that the navigation map is no longer empty, set the
```

```
movement target.
    set_movement_target(movement_target_position)

func set_movement_target(movement_target: Vector2):
    navigation_agent.target_position = movement_target

func _physics_process(delta):
    if navigation_agent.is_navigation_finished():
        return

    var current_agent_position: Vector2 = global_position
    var next_path_position: Vector2 =
navigation_agent.get_next_path_position()

    velocity =
current_agent_position.direction_to(next_path_position) *
movement_speed
    move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private NavigationAgent2D _navigationAgent;

    private float _movementSpeed = 200.0f;
    private Vector2 _movementTargetPosition = new
Vector2(70.0f, 226.0f);

    public Vector2 MovementTarget
    {
        get { return _navigationAgent.TargetPosition; }
        set { _navigationAgent.TargetPosition = value; }
    }

    public override void _Ready()
    {
        base._Ready();

        _navigationAgent = GetNode<NavigationAgent2D>
```

```
("NavigationAgent2D");

        // These values need to be adjusted for the actor's
speed
        // and the navigation layout.
        _navigationAgent.PathDesiredDistance = 4.0f;
        _navigationAgent.TargetDesiredDistance = 4.0f;

        // Make sure to not await during _Ready.
        Callable.From(ActorSetup).CallDeferred();
    }

    public override void _PhysicsProcess(double delta)
    {
        base._PhysicsProcess(delta);

        if (_navigationAgent.IsNavigationFinished())
        {
            return;
        }

        Vector2 currentAgentPosition = GlobalTransform.Origin;
        Vector2 nextPathPosition =
_navigationAgent.GetNextPathPosition();

        Velocity =
currentAgentPosition.DirectionTo(nextPathPosition) *
_movementSpeed;
        MoveAndSlide();
    }

    private async void ActorSetup()
    {
        // Wait for the first physics frame so the
NavigationServer can sync.
        await ToSignal(GetTree(),
SceneTree.SignalName.PhysicsFrame);

        // Now that the navigation map is no longer empty, set
the movement target.
        MovementTarget = _movementTargetPosition;
        }
}
```

Note

On the first frame the NavigationServer map has not synchronized region data and any path query will return empty. Wait for the NavigationServer synchronization by awaiting one frame in the script.

3D navigation overview

Godot provides multiple objects, classes and servers to facilitate grid-based or mesh-based navigation and pathfinding for 2D and 3D games. The following section provides a quick overview over all available navigation related objects in Godot for 3D scenes and their primary use.

Godot provides the following objects and classes for 3D navigation:

- Astar3D

  `Astar3D` objects provide an option to find the shortest path in a graph of weighted **points**.

  The AStar3D class is best suited for cell-based 3D gameplay that does not require actors to reach any possible position within an area but only predefined, distinct positions.

- NavigationServer3D

  `NavigationServer3D` provides a powerful server API to find the shortest path between two positions on an area defined by a navigation mesh.

  The NavigationServer is best suited for 3D realtime gameplay that does require actors to reach any possible position within a navigation mesh defined area. Mesh-based navigation scales well with large game worlds as a large area can often be defined with a single polygon when it would require many, many grid cells.

  The NavigationServer holds different navigation maps that each consist of regions that hold navigation mesh data. Agents can be placed on a map for avoidance calculation.

RIDs are used to reference internal maps, regions, and agents when communicating with the server.

The following NavigationServer RID types are available.

- NavMap RID
  Reference to a specific navigation map that holds regions and agents. The map will attempt to join the navigation meshes of the regions by proximity. The map will synchronize regions and agents each physics frame.

- NavRegion RID
  Reference to a specific navigation region that can hold navigation mesh data. The region can be enabled / disabled or the use restricted with a navigation layer bitmask.

- NavLink RID
  Reference to a specific navigation link that connects two navigation mesh positions over arbitrary distances.

- NavAgent RID
  Reference to a specific avoidance agent. The avoidance is defined by a radius value.

- NavObstacle RID
  Reference to a specific avoidance obstacle used to affect and constrain the avoidance velocity of agents.

The following scene tree nodes are available as helpers to work with the NavigationServer3D API.

- NavigationRegion3D Node
  A Node that holds a Navigation Mesh resource that defines a navigation mesh for the NavigationServer3D.

    - The region can be enabled / disabled.
    - The use in pathfinding can be further restricted through the `navigation_layers` bitmask.
    - The NavigationServer3D will join the navigation meshes of regions by proximity for a combined navigation mesh.
- NavigationLink3D Node

  A Node that connects two positions on navigation meshes over arbitrary distances for pathfinding.

    - The link can be enabled / disabled.
    - The link can be made one-way or bidirectional.
    - The use in pathfinding can be further restricted through the `navigation_layers` bitmask.

  Links tell the pathfinding that a connection exists and at what cost. The actual agent handling and movement needs to happen in custom scripts.

- NavigationAgent3D Node

  A helper Node used to facilitate common NavigationServer3D API calls for pathfinding and avoidance. Use this Node with a Node3D inheriting parent Node.

- NavigationObstacle3D Node

  A Node that can be used to affect and constrain the avoidance velocity of avoidance enabled agents. This Node does NOT affect the pathfinding of agents. You need to change the navigation meshes for that instead.

The 3D navigation meshes are defined with the following resources:

- NavigationMesh Resource

  A resource that holds 3D navigation mesh data. It provides 3D geometry baking options to define navigation areas inside the Editor as well as at runtime.

    - The NavigationRegion3D Node uses this resource to define its navigation area.

  - The NavigationServer3D uses this resource to update the navigation mesh of individual regions.
  - The GridMap Editor uses this resource when specific navigation meshes are defined for each grid cell.

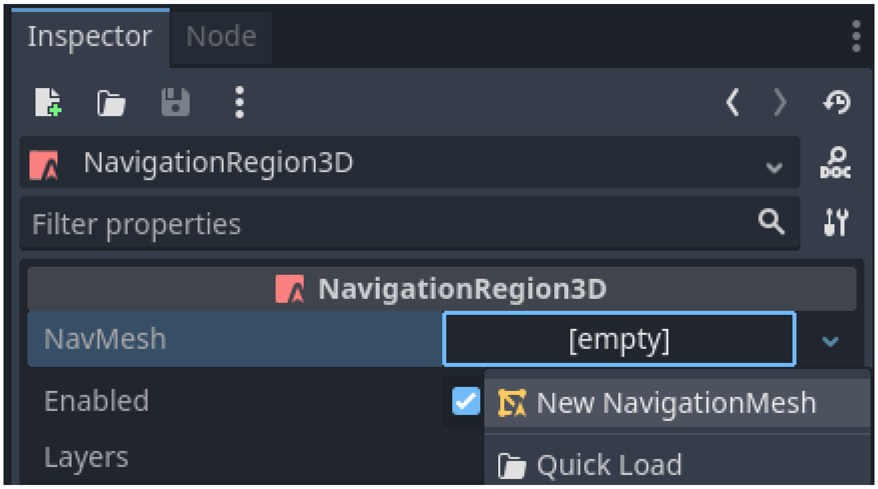
See also

You can see how 3D navigation works in action using the 3D Navigation demo project [https://github.com/godotengine/godot-demo-projects/tree/master/3d/navigation].

Setup for 3D scene

The following steps show a basic setup for minimal viable navigation in 3D. It uses the NavigationServer3D and a NavigationAgent3D for path movement.

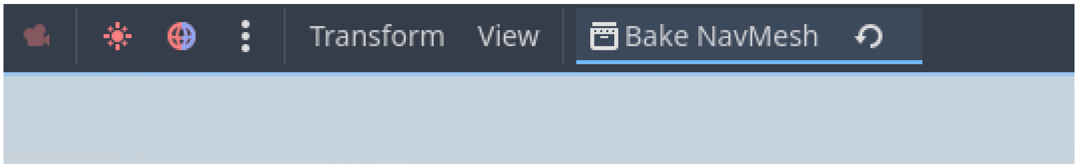
1. Add a NavigationRegion3D Node to the scene.
2. Click on the region node and add a new NavigationMesh Resource to the region node.


Inspector
Node
NavigationRegion3D
Filter properties
NavigationRegion3D
NavMesh
[empty]
Enabled
Layers
New NavigationMesh
Quick Load

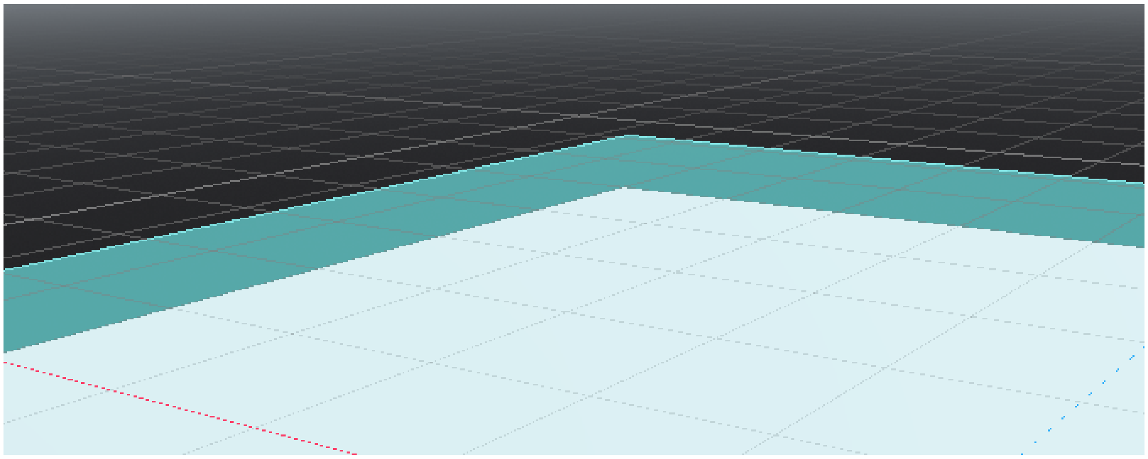
3. Add a new MeshInstance3D node as a child of the region node.

4. Select the MeshInstance3D node and add a new PlaneMesh and increase the xy size to 10.

5. Select the region node again and press the "Bake Navmesh" button on the top bar.

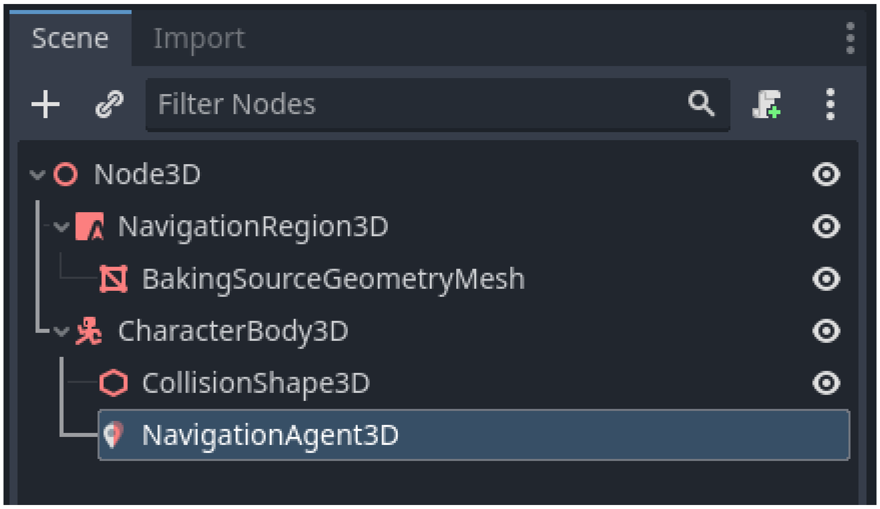


6. Now a transparent navigation mesh appears that hovers some distance on top of the PlaneMesh.



7. Add a CharacterBody3D node in the scene with a basic collision shape and some mesh for visuals.

8. Add a NavigationAgent3D node below the character node.


Scene
Import
Filter Nodes
Node3D
NavigationRegion3D
BakingSourceGeometryMesh
CharacterBody3D
CollisionShape3D
NavigationAgent3D

9. Add a script to the CharacterBody3D node with the following content. We make sure to set a movement target after the scene has fully loaded and the NavigationServer had time to sync. Also, add a Camera3D and some light and environment to see something.

GDScript

```
extends CharacterBody3D

var movement_speed: float = 2.0
var movement_target_position: Vector3 = Vector3(-3.0,0.0,2.0)

@onready var navigation_agent: NavigationAgent3D =
$NavigationAgent3D

func _ready():
    # These values need to be adjusted for the actor's speed
    # and the navigation layout.
    navigation_agent.path_desired_distance = 0.5
    navigation_agent.target_desired_distance = 0.5

    # Make sure to not await during _ready.
    actor_setup.call_deferred()

func actor_setup():
    # Wait for the first physics frame so the NavigationServer
can sync.
```

```
    await get_tree().physics_frame

    # Now that the navigation map is no longer empty, set the movement target.
    set_movement_target(movement_target_position)

func set_movement_target(movement_target: Vector3):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    if navigation_agent.is_navigation_finished():
        return

    var current_agent_position: Vector3 = global_position
    var next_path_position: Vector3 = navigation_agent.get_next_path_position()

    velocity = current_agent_position.direction_to(next_path_position) * movement_speed
    move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody3D : CharacterBody3D
{
    private NavigationAgent3D _navigationAgent;

    private float _movementSpeed = 2.0f;
    private Vector3 _movementTargetPosition = new Vector3(-3.0f, 0.0f, 2.0f);

    public Vector3 MovementTarget
    {
        get { return _navigationAgent.TargetPosition; }
        set { _navigationAgent.TargetPosition = value; }
    }

    public override void _Ready()
    {
        base._Ready();

        _navigationAgent = GetNode<NavigationAgent3D>
("NavigationAgent3D");

        // These values need to be adjusted for the actor's
speed
        // and the navigation layout.
        _navigationAgent.PathDesiredDistance = 0.5f;
        _navigationAgent.TargetDesiredDistance = 0.5f;

        // Make sure to not await during _Ready.
        Callable.From(ActorSetup).CallDeferred();
    }

    public override void _PhysicsProcess(double delta)
    {
        base._PhysicsProcess(delta);

        if (_navigationAgent.IsNavigationFinished())
        {
            return;
        }

        Vector3 currentAgentPosition = GlobalTransform.Origin;
        Vector3 nextPathPosition =
_navigationAgent.GetNextPathPosition();

        Velocity =
currentAgentPosition.DirectionTo(nextPathPosition) *
_movementSpeed;
        MoveAndSlide();
    }

    private async void ActorSetup()
    {
        // Wait for the first physics frame so the
NavigationServer can sync.
        await ToSignal(GetTree(),
SceneTree.SignalName.PhysicsFrame);

        // Now that the navigation map is no longer empty, set
the movement target.
```

```
        MovementTarget = _movementTargetPosition;
    }
}
```

Note

On the first frame the NavigationServer map has not synchronized region data and any path query will return empty. Wait for the NavigationServer synchronization by awaiting one frame in the script.

Using NavigationServer

2D and 3D version of the NavigationServer are available as NavigationServer2D and NavigationServer3D respectively.

Both 2D and 3D use the same NavigationServer with NavigationServer3D being the primary server. The NavigationServer2D is a frontend that converts 2D positions into 3D positions and back. Hence it is entirely possible (if not a little cumbersome) to exclusively use the NavigationServer3D API for 2D navigation.

Communicating with the NavigationServer

To work with the NavigationServer means to prepare parameters for a **query** that can be sent to the NavigationServer for updates or requesting data.

To reference the internal NavigationServer objects like maps, regions and agents RIDs are used as identification numbers. Every navigation related node in the scene tree has a function that returns the RID for this node.

Threading and Synchronization

The NavigationServer does not update every change immediately but waits until the end of the **physics frame** to synchronize all the changes together.

Waiting for synchronization is required to apply changes to all maps, regions and agents. Synchronization is done because some updates like a recalculation of the entire navigation map are very expensive and require updated data from all other objects. Also the NavigationServer uses a **threadpool** by default for some functionality like avoidance calculation between agents.

Waiting is not required for most `get()` functions that only request data from the NavigationServer without making changes. Note that not all data will account for changes made in the same frame. E.g. if an avoidance agent changed the navigation map this frame the `agent_get_map()` function will still return the old map before the synchronization. The exception to this are nodes that store their values internally before sending the update to the NavigationServer. When a getter on a node is used for a value that was updated in the same frame it will return the already updated value stored on the node.

The NavigationServer is **thread-safe** as it places all API calls that want to make changes in a queue to be executed in the synchronization phase. Synchronization for the NavigationServer happens in the middle of the physics frame after scene input from scripts and nodes are all done.

Note

The important takeaway is that most NavigationServer changes take effect after the next physics frame and not immediately. This includes all changes made by navigation related nodes in the scene tree or through scripts.

Note

All setters and delete functions require synchronization.

2D and 3D NavigationServer differences

NavigationServer2D and NavigationServer3D are equivalent in functionality for their dimension and both use the same NavigationServer behind the scene.

Strictly technical a NavigationServer2D is a myth. The NavigationServer2D is a frontend to facilitate conversions of `Vector2(x, y)` to `Vector3(x, 0.0, z)` and back for the NavigationServer3D API. 2D uses a flat 3D mesh pathfinding and the NavigationServer2D facilitates the conversions. When a guide uses just NavigationServer without the 2D or 3D suffix it usually works for both servers by exchange `Vector2(x, y)` with `Vector3(x, 0.0, z)` or reverse.

Technically it is possible to use the tools for creating navigation meshes in one dimension for the other dimension, e.g. baking a 2D navigation mesh with the 3D NavigationMesh when using flat 3D source geometry or creating 3D flat navigation meshes with the polygon outline draw tools of NavigationRegion2D and NavigationPolygons.

Any RID created with the NavigationServer2D API works on the NavigationServer3D API as well and both 2D and 3D avoidance agents can exist on the same map.

Note

Regions created in 2D and 3D will merge their navigation meshes when placed on the same map and merge conditions apply. The NavigationServer does not discriminate between NavigationRegion2D and NavigationRegion3D nodes as both are regions on the server. By default those nodes register on different navigation maps so this merge can only happen when maps are changed manually e.g. with scripts.

Actors with avoidance enabled will avoid both 2D and 3D avoidance agents when placed on the same map.

Warning

It is not possible to use NavigationServer2D while disabling 3D on a Godot custom build.

Waiting for synchronization

At the start of the game, a new scene or procedural navigation changes any path query to a NavigationServer will return empty or wrong.

The navigation map is still empty or not updated at this point. All nodes from the scene tree need to first upload their navigation related data to the NavigationServer. Each added or changed map, region or agent need to be registered with the NavigationServer. Afterward the NavigationServer requires a **physics frame** for synchronization to update the maps, regions and agents.

One workaround is to make a deferred call to a custom setup function (so all nodes are ready). The setup function makes all the navigation changes, e.g. adding procedural stuff. Afterwards the function waits for the next physics frame before continuing with path queries.

GDScript

```
extends Node3D
```

```
func _ready():
    # Use call deferred to make sure the entire scene tree
nodes are setup
    # else await on 'physics_frame' in a _ready() might get
stuck.
    custom_setup.call_deferred()

func custom_setup():

    # Create a new navigation map.
    var map: RID = NavigationServer3D.map_create()
    NavigationServer3D.map_set_up(map, Vector3.UP)
    NavigationServer3D.map_set_active(map, true)

    # Create a new navigation region and add it to the map.
    var region: RID = NavigationServer3D.region_create()
    NavigationServer3D.region_set_transform(region,
Transform3D())
    NavigationServer3D.region_set_map(region, map)

    # Create a procedural navigation mesh for the region.
    var new_navigation_mesh: NavigationMesh =
NavigationMesh.new()
    var vertices: PackedVector3Array = PackedVector3Array([
        Vector3(0, 0, 0),
        Vector3(9.0, 0, 0),
        Vector3(0, 0, 9.0)
    ])
    new_navigation_mesh.set_vertices(vertices)
    var polygon: PackedInt32Array = PackedInt32Array([0, 1, 2])
    new_navigation_mesh.add_polygon(polygon)
    NavigationServer3D.region_set_navigation_mesh(region,
new_navigation_mesh)

    # Wait for NavigationServer sync to adapt to made changes.
    await get_tree().physics_frame

    # Query the path from the navigation server.
    var start_position: Vector3 = Vector3(0.1, 0.0, 0.1)
    var target_position: Vector3 = Vector3(1.0, 0.0, 1.0)
    var optimize_path: bool = true

    var path: PackedVector3Array =
NavigationServer3D.map_get_path(
        map,
        start_position,
        target_position,
        optimize_path
    )

    print("Found a path!")
    print(path)
```

C#

```
using Godot;

public partial class MyNode3D : Node3D
{
    public override void _Ready()
    {
        // Use call deferred to make sure the entire scene tree
nodes are setup
        // else await on 'physics_frame' in a _Ready() might
get stuck.
        CallDeferred(MethodName.CustomSetup);
    }

    private async void CustomSetup()
    {
        // Create a new navigation map.
        Rid map = NavigationServer3D.MapCreate();
        NavigationServer3D.MapSetUp(map, Vector3.Up);
        NavigationServer3D.MapSetActive(map, true);

        // Create a new navigation region and add it to the
map.
        Rid region = NavigationServer3D.RegionCreate();
        NavigationServer3D.RegionSetTransform(region,
Transform3D.Identity);
        NavigationServer3D.RegionSetMap(region, map);

        // Create a procedural navigation mesh for the region.
        var newNavigationMesh = new NavigationMesh()
        {
            Vertices = new[]
            {
                new Vector3(0.0f, 0.0f, 0.0f),
                new Vector3(9.0f, 0.0f, 0.0f),
                new Vector3(0.0f, 0.0f, 9.0f),
            },
        };
        int[] polygon = new[] { 0, 1, 2 };
        newNavigationMesh.AddPolygon(polygon);
        NavigationServer3D.RegionSetNavigationMesh(region,
newNavigationMesh);

        // Wait for NavigationServer sync to adapt to made
changes.
        await ToSignal(GetTree(),
SceneTree.SignalName.PhysicsFrame);

        // Query the path from the navigation server.
        var startPosition = new Vector3(0.1f, 0.0f, 0.1f);
        var targetPosition = new Vector3(1.0f, 0.0f, 1.0f);

        Vector3[] path = NavigationServer3D.MapGetPath(map,
startPosition, targetPosition, optimize: true);

        GD.Print("Found a path!");
        GD.Print((Variant)path);
    }
}
```

Server Avoidance Callbacks

If RVO avoidance agents are registered for avoidance callbacks the NavigationServer dispatches their `velocity_computed` signals just before the PhysicsServer synchronization.

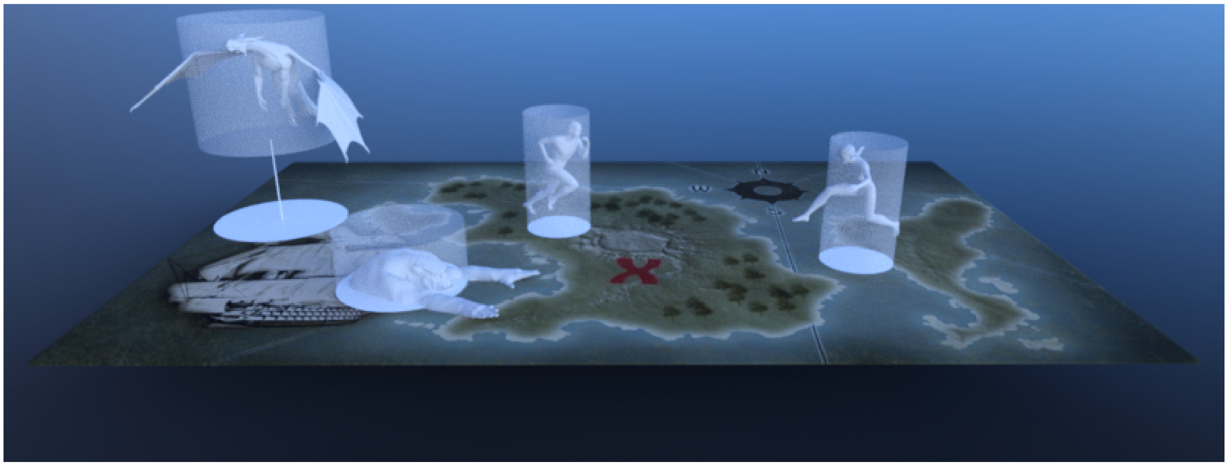
To learn more about NavigationAgents see Using NavigationAgents.

The simplified order of execution for NavigationAgents that use avoidance:

- physics frame starts.
- `_physics_process(delta)`.
- `velocity` property is set on NavigationAgent Node.
- Agent sends velocity and position to NavigationServer.
- NavigationServer waits for synchronization.
- NavigationServer synchronizes and computes avoidance velocities for all registered avoidance agents.
- NavigationServer sends safe velocity vector with signals for each registered avoidance agents.
- Agents receive the signal and move their parent e.g. with `move_and_slide` or `linear_velocity`.
- PhysicsServer synchronizes.
- physics frame ends.

Therefore moving a physicsbody actor in the callback function with the safe velocity is perfectly thread- and physics-safe as all happens inside the same physics frame before the PhysicsServer commits to changes and does its own calculations.

Using NavigationMaps



A NavigationMap is an abstract navigation world on the NavigationServer identified by a NavigationServer RID.

A map can hold and connect a near infinite number of navigation regions with navigation meshes to build the traversable areas of a game world for pathfinding.

A map can contain avoidance agents. Collision avoidance will be calculated based on the agents present in the map.

Note

Different NavigationMaps are completely isolated from each other but navigation regions and avoidance agents can switch between different maps. Switches will become effective on NavigationServer synchronization.

Default navigation maps

By default Godot creates a navigation map for each World2D and World3D of the root viewport.

The 2D default navigation map RID can be obtained with `get_world_2d().get_navigation_map()` from any Node2D inheriting Node.

The 3D default navigation map RID can be obtained with `get_world_3d().get_navigation_map()` from any Node3D inheriting Node.

2D GDScript

```
extends Node2D

func _ready() -> void:
    var default_navigation_map_rid: RID =
get_world_2d().get_navigation_map()
```

2D C#

```
public partial class MyNode2D : Node2D
{
    public override void _Ready()
    {
        Rid defaultNavigationMapRid =
GetWorld2D().NavigationMap;
    }
}
```

3D GDScript

```
extends Node3D

func _ready() -> void:
    var default_navigation_map_rid: RID =
get_world_3d().get_navigation_map()
```

3D C#

```
public partial class MyNode3D : Node3D
{
    public override void _Ready()
    {
        Rid defaultNavigationMapRid =
GetWorld3D().NavigationMap;
    }
}
```

Creating new navigation maps

The NavigationServer can create and support as many navigation maps as required for specific gameplay. Additional navigation maps are created and handled by using the NavigationServer API directly e.g. to support different avoidance agent or actor locomotion types.

For example uses of different navigation maps see Support different actor types and Support different actor locomotion.

Each navigation map individually synchronizes queued changes to its navigation regions and avoidance agents. A navigation map that has not received changes will consume little to no processing time. Navigation regions and avoidance agents can only be part of a single navigation map but they can switch map at any time.

Note

A navigation map switch will take effect only after the next NavigationServer synchronization.

2D GDScript

```
extends Node2D

func _ready() -> void:
    var new_navigation_map: RID =
NavigationServer2D.map_create()
    NavigationServer2D.map_set_active(new_navigation_map, true)
```

2D C#

```
public partial class MyNode2D : Node2D
{
    public override void _Ready()
    {
        Rid newNavigationMap = NavigationServer2D.MapCreate();
        NavigationServer2D.MapSetActive(newNavigationMap,
true);
    }
}
```

3D GDScript

```
extends Node3D

func _ready() -> void:
    var new_navigation_map: RID =
NavigationServer3D.map_create()
    NavigationServer3D.map_set_active(new_navigation_map, true)
```

3D C#

```
public partial class MyNode3D : Node3D
{
    public override void _Ready()
    {
        Rid newNavigationMap = NavigationServer3D.MapCreate();
        NavigationServer3D.MapSetActive(newNavigationMap,
true);
    }
}
```

Note

There is no difference between navigation maps created with the NavigationServer2D API or the NavigationServer3D API.

Using NavigationRegions

NavigationRegions are the visual Node representation of a **region** of the navigation **map** on the NavigationServer. Each NavigationRegion node holds a resource for the navigation mesh data.

Both 2D and 3D version are available as NavigationRegion2D and NavigationRegion3D respectively.

Individual NavigationRegions upload their 2D NavigationPolygon or 3D NavigationMesh resource data to the NavigationServer. The NavigationServer map turns this information into a combined navigation map for pathfinding.

To create a navigation region using the scene tree add a `NavigationRegion2D` or `NavigationRegion3D` node to the scene. All regions require a navigation mesh resource to function. See Using navigation meshes to learn how to create and apply navigation meshes.

NavigationRegions will automatically push `global_transform` changes to the region on the NavigationServer which makes them suitable for moving platforms. The NavigationServer will attempt to connect the navigation meshes of individual regions when they are close enough. For more details see Connecting navigation meshes. To connect NavigationRegions over arbitrary distances see Using NavigationLinks to learn how to create and use `NavigationLinks`.

Warning

While changing the transform of a NavigationRegion node does update the region position on the NavigationServer, changing the scale does not. A navigation mesh resource has no scale and needs to be fully updated when source geometry changes scale.

Regions can be enabled / disabled and if disabled will not contribute to future pathfinding queries.

Note

Existing paths will not be automatically updated when a region gets enabled / disabled.

Creating new navigation regions

New NavigationRegion nodes will automatically register to the default world navigation map for their 2D/3D dimension.

The region RID can then be obtained from NavigationRegion Nodes with `get_rid()`.

2D GDScript

```
extends NavigationRegion2D

var navigationserver_region_rid: RID = get_rid()
```

2D C#

```
public partial class MyNavigationRegion2D : NavigationRegion2D
{
    public override void _Ready()
    {
        Rid navigationServerRegionRid = GetRid();
    }
}
```

3D GDScript

```
extends NavigationRegion3D

var navigationserver_region_rid: RID = get_rid()
```

3D C#

```
public partial class MyNavigationRegion3D : NavigationRegion3D
{
    public override void _Ready()
    {
        Rid navigationServerRegionRid = GetRid();
    }
}
```

New regions can also be created with the NavigationServer API and added to any existing map.

If regions are created with the NavigationServer API directly they need to be assigned a navigation map manually.

2D GDScript

```
extends Node2D

func _ready() -> void:
    var new_region_rid: RID =
NavigationServer2D.region_create()
    var default_map_rid: RID =
get_world_2d().get_navigation_map()
    NavigationServer2D.region_set_map(new_region_rid,
default_map_rid)
```

2D C#

```
public partial class MyNode2D : Node2D
{
    public override void _Ready()
    {
        Rid newRegionRid = NavigationServer2D.RegionCreate();
        Rid defaultMapRid = GetWorld2D().NavigationMap;
        NavigationServer2D.RegionSetMap(newRegionRid,
defaultMapRid);
    }
}
```

3D GDScript

```
extends Node3D

func _ready() -> void:
    var new_region_rid: RID =
NavigationServer3D.region_create()
    var default_map_rid: RID =
get_world_3d().get_navigation_map()
    NavigationServer3D.region_set_map(new_region_rid,
default_map_rid)
```

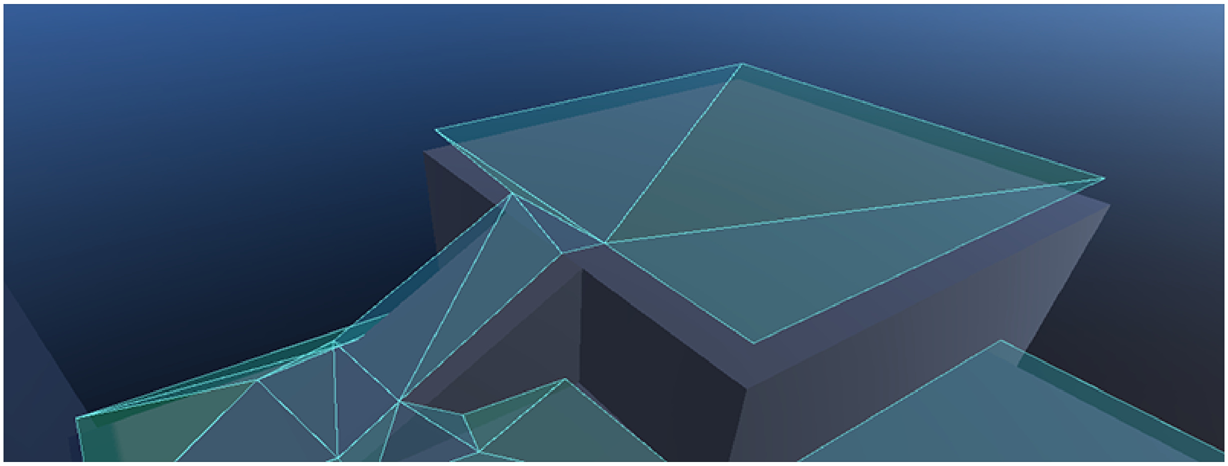
3D C#

```
public partial class MyNode3D : Node3D
{
    public override void _Ready()
    {
        Rid newRegionRid = NavigationServer3D.RegionCreate();
        Rid defaultMapRid = GetWorld3D().NavigationMap;
        NavigationServer3D.RegionSetMap(newRegionRid,
defaultMapRid);
    }
}
```

Note

Navigation regions can only be assigned to a single navigation map. If an existing region is assigned to a new navigation map it will leave the old map.

Using navigation meshes

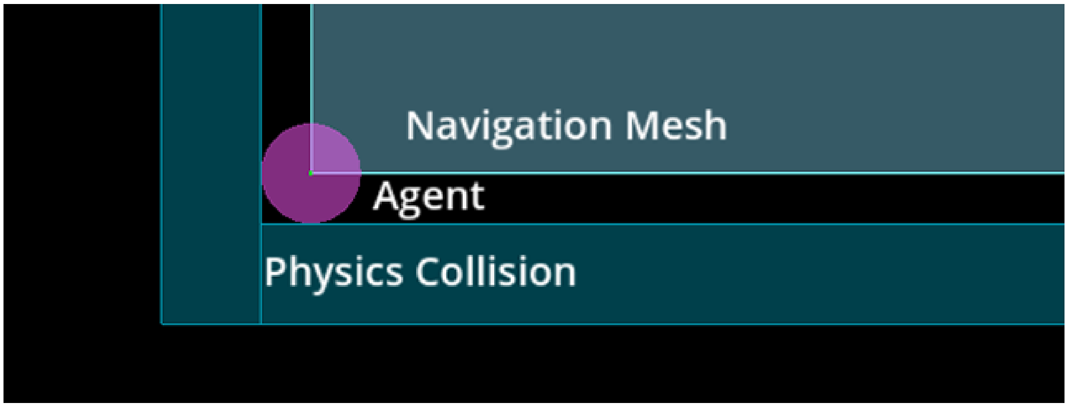


2D and 3D versions of the navigation mesh are available as NavigationPolygon and NavigationMesh respectively.

Note

A navigation mesh only describes a traversable area for an agent's center position. Any radius values an agent may have are ignored. If you want pathfinding to account for an agent's (collision) size you need to shrink the navigation mesh accordingly.

Navigation works independently from other engine parts like rendering or physics. Navigation meshes are the only things considered when doing pathfinding, e.g. visuals and collision shapes for example are completely ignored by the navigation system. If you need to take other data (like visuals for example) into account when doing pathfinding, you need to adapt your navigation meshes accordingly. The process of factoring in navigation restrictions in navigation meshes is commonly referred to as navigation mesh baking.

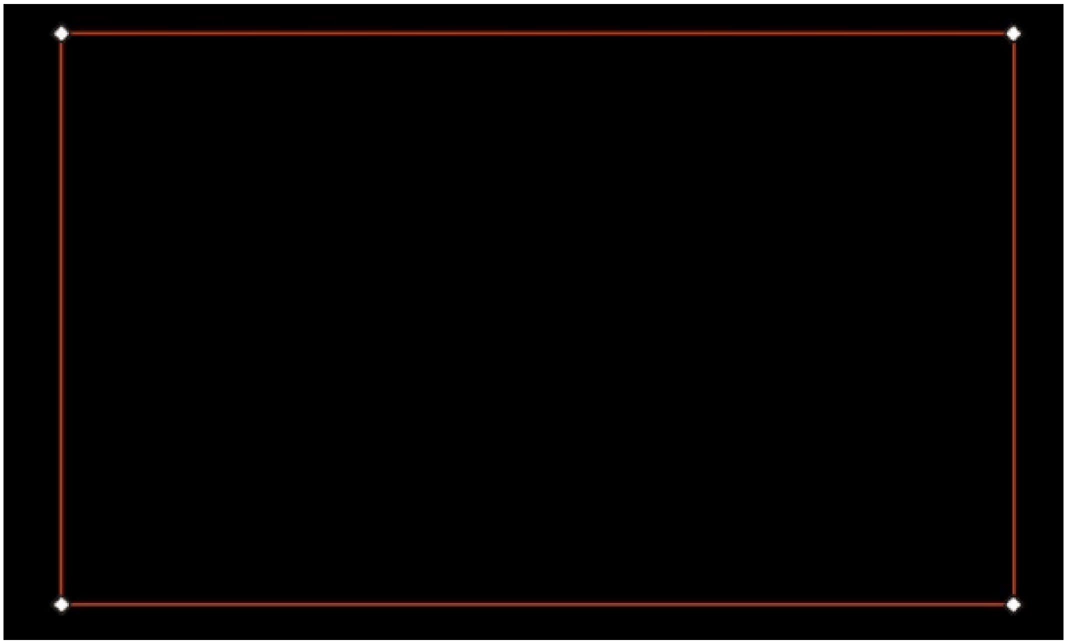


A navigation mesh describes a surface that an agent can stand on safely with its center compared to physics shapes that describe outer collision bounds.

If you experience clipping or collision problems while following navigation paths, always remember that you need to tell the navigation system what your intentions are through an appropriate navigation mesh. By itself the navigation system will never know "this is a tree / rock / wall collision shape or visual mesh" because it only knows that "here I was told I can path safely because it is on a navigation mesh".

Navigation mesh baking can be done either by using a NavigationRegion2D or NavigationRegion3D, or by using the NavigationServer2D and NavigationServer3D API directly.

Baking a navigation mesh with a NavigationRegion



Baking a navigation mesh with agent radius offset from geometry.

The navigation mesh baking is made more accessible with the NavigationRegion node. When baking with a NavigationRegion node, the individual parsing, baking, and region update steps are all combined into one function.

The nodes are available in 2D and 3D as NavigationRegion2D and NavigationRegion3D respectively.

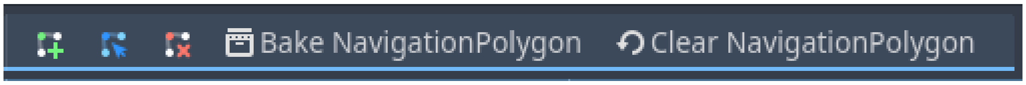
Tip

The navigation mesh `source_geometry_mode` can be switched to parse specific node group names so nodes that should be baked can be placed anywhere in the scene.

Baking with a NavigationRegion2D

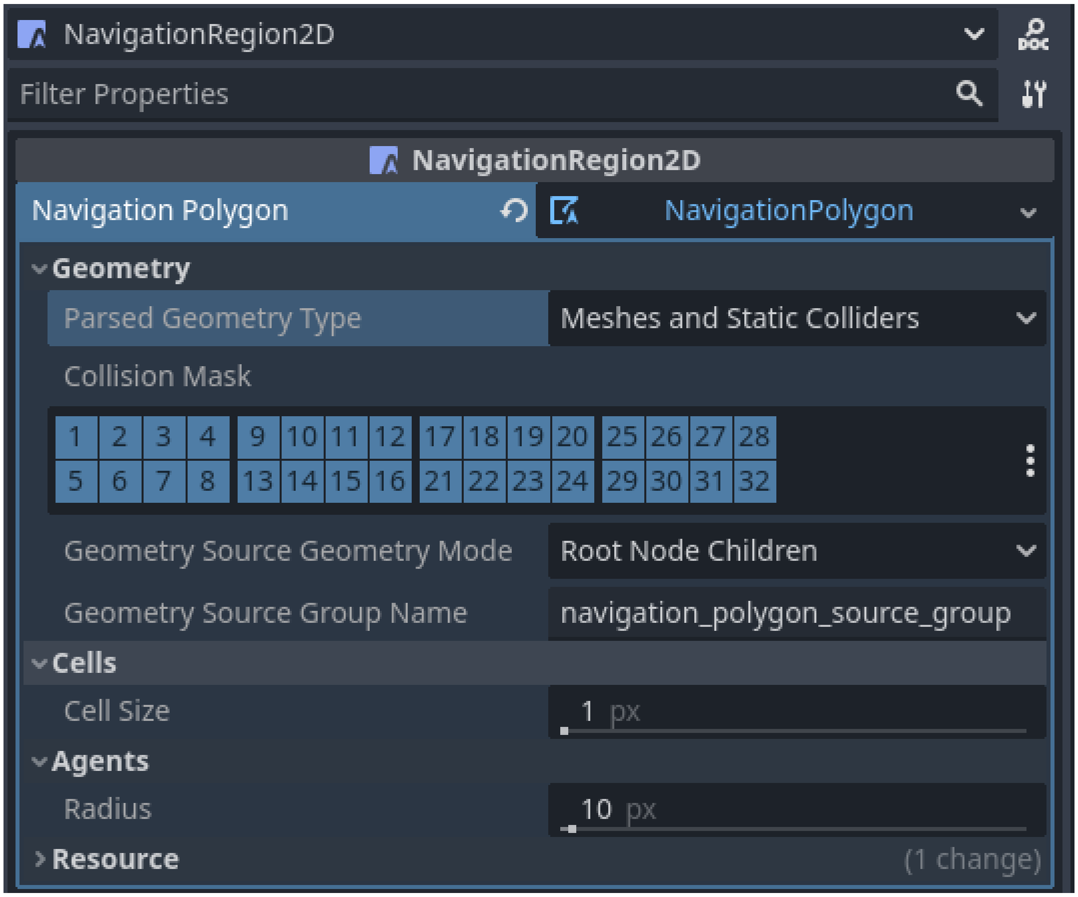
When a NavigationRegion2D node is selected in the Editor, bake options as well as polygon draw tools appear in the top bar of the

Editor.


Bake NavigationPolygon
Clear NavigationPolygon

In order for the region to work a NavigationPolygon resource needs to be added.

The properties to parse and bake a navigation mesh are then part of the used resource and can be found in the resource Inspector.


NavigationRegion2D
Filter Properties
NavigationRegion2D

Navigation Polygon: NavigationPolygon

Geometry

Parsed Geometry Type: Meshes and Static Colliders

Collision Mask: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Geometry Source Geometry Mode: Root Node Children

Geometry Source Group Name: navigation_polygon_source_group

Cells

Cell Size: 1 px

Agents

Radius: 10 px

Resource (1 change)

The result of the source geometry parsing can be influenced with the following properties.

- The `parsed_geometry_type` that filters if visual objects or physics objects or both should be parsed from the SceneTree. For more details on what objects are parsed and how, see the section about parsing source geometry below.

- The `collision_mask` filters which physics collision objects are included when the `parsed_geometry_type` includes static colliders.
- The `source_geometry_mode` that defines on which node(s) to start the parsing, and how to traverse the SceneTree.
- The `source_geometry_group_name` is used when only a certain node group should be parsed. Depends on the selected `source_geometry_mode`.

With the source geometry added, the result of the baking can be controlled with the following properties.

- The `cell_size` sets the rasterization grid size and should match the navigation map size.
- The `agent_radius` shrinks the baked navigation mesh to have enough margin for the agent (collision) size.

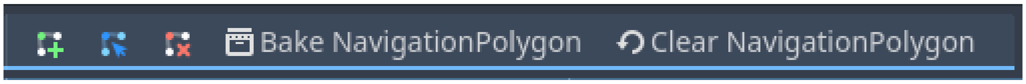
The NavigationRegion2D baking can also be used at runtime with scripts.

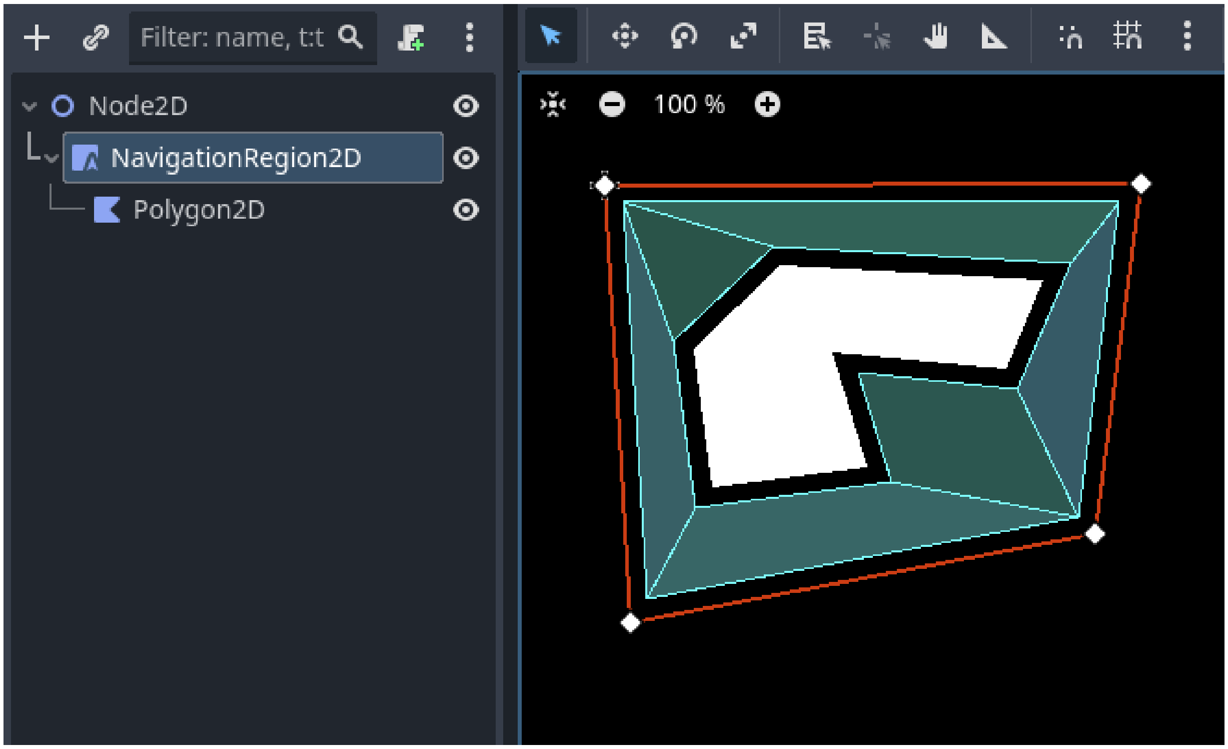
GDScript

```
var on_thread: bool = true
bake_navigation_polygon(on_thread)
```

To quickly test the 2D baking with default settings:

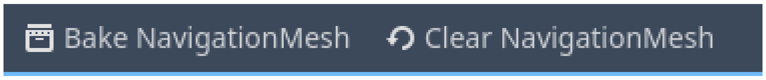
- Add a NavigationRegion2D.
- Add a NavigationPolygon resource to the NavigationRegion2D.
- Add a Polygon2D below the NavigationRegion2D.
- Draw 1 NavigationPolygon outline with the selected NavigationRegion2D draw tool.
- Draw 1 Polygon2D outline inside the NavigationPolygon outline with the selected Polygon2D draw tool.
- Hit the Editor bake button and a navigation mesh should appear.


Bake NavigationPolygon
Clear NavigationPolygon


Filter: name, t:t
Node2D
NavigationRegion2D
Polygon2D
100 %

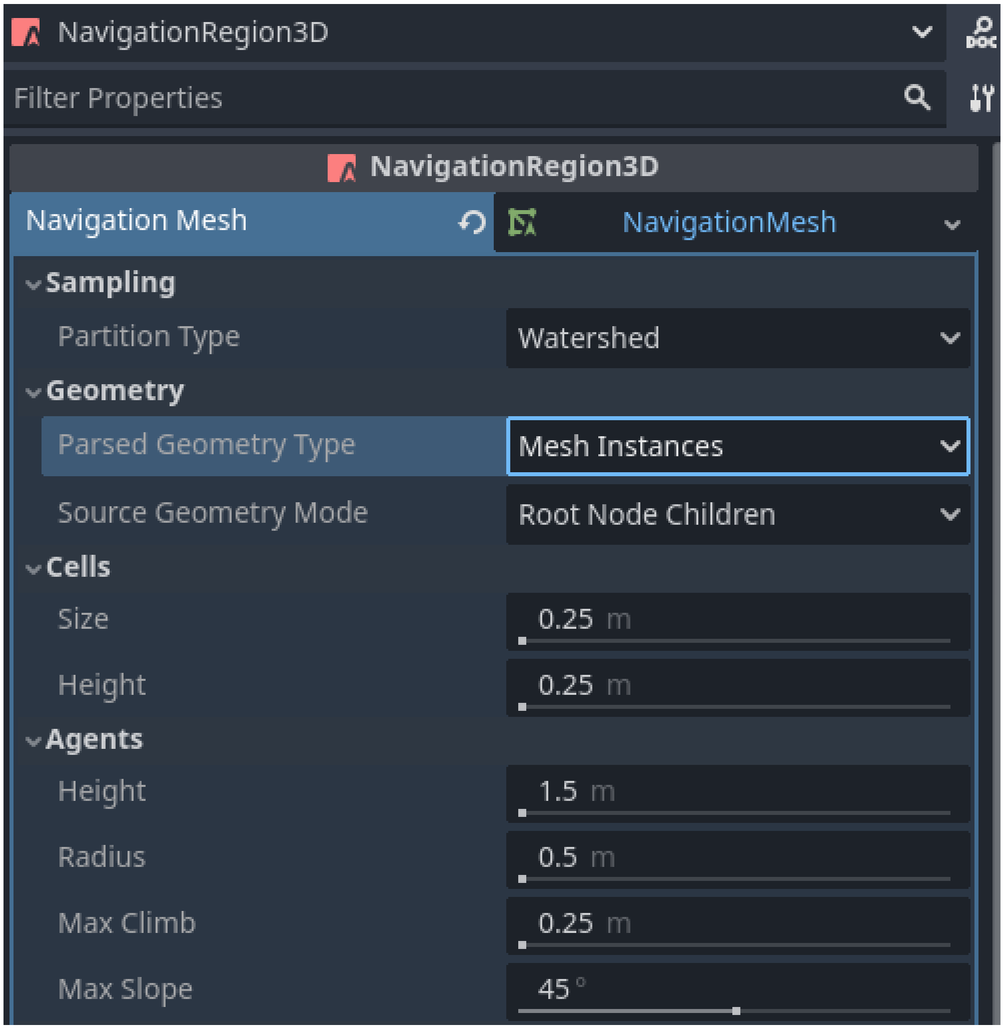
Baking with a NavigationRegion3D

When a NavigationRegion3D node is selected in the Editor, bake options appear in the top bar of the Editor.


Bake NavigationMesh
Clear NavigationMesh

In order for the region to work a NavigationMesh resource needs to be added.

The properties to parse and bake a navigation mesh are then part of the used resource and can be found in the resource Inspector.



Navigation Mesh: NavigationMesh

Sampling

Partition Type: Watershed

Geometry

Parsed Geometry Type: Mesh Instances

Source Geometry Mode: Root Node Children

Cells

Size: 0.25 m

Height: 0.25 m

Agents

Height: 1.5 m

Radius: 0.5 m

Max Climb: 0.25 m

Max Slope: 45 °

The result of the source geometry parsing can be influenced with the following properties.

- The `parsed_geometry_type` that filters if visual objects or physics objects or both should be parsed from the SceneTree. For more details on what objects are parsed and how, see the section about parsing source geometry below.
- The `collision_mask` filters which physics collision objects are included when the `parsed_geometry_type` includes static colliders.
- The `source_geometry_mode` that defines on which node(s) to start the parsing, and how to traverse the SceneTree.

- The `source_geometry_group_name` is used when only a certain node group should be parsed. Depends on the selected `source_geometry_mode`.

With the source geometry added, the result of the baking can be controlled with the following properties.

- The `cell_size` and `cell_height` sets the rasterization voxel grid size and should match the navigation map size.
- The `agent_radius` shrinks the baked navigation mesh to have enough margin for the agent (collision) size.
- The `agent_height` excludes areas from the navigation mesh where the agent is too tall to fit in.
- The `agent_max_climb` and `agent_max_slope` removes areas where the height difference between neighboring voxels is too large, or where their surface is too steep.

Warning

A too small `cell_size` or `cell_height` can create so many voxels that it has the potential to freeze the game or even crash.

The NavigationRegion3D baking can also be used at runtime with scripts.

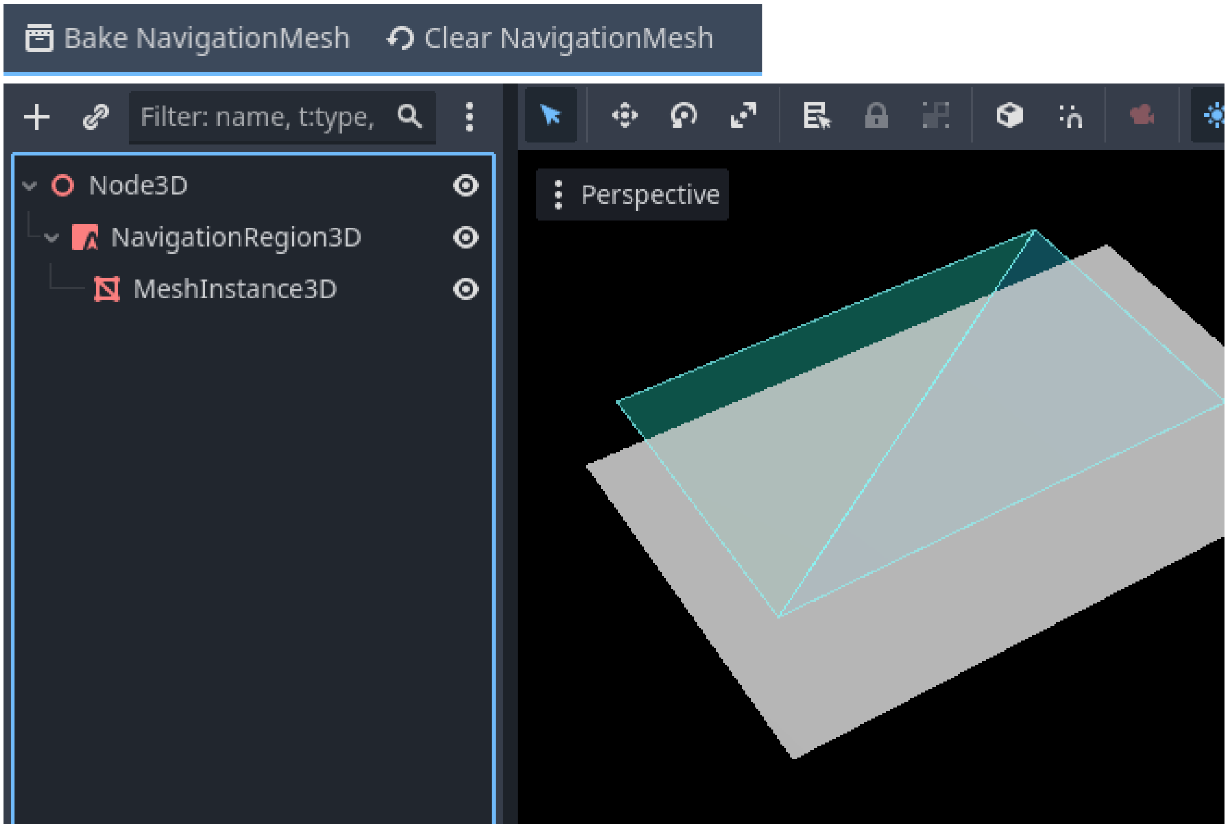
GDScript

```
var on_thread: bool = true
bake_navigation_mesh(on_thread)
```

To quickly test the 3D baking with default settings:

- Add a NavigationRegion3D.
- Add a NavigationMesh resource to the NavigationRegion3D.
- Add a MeshInstance3D below the NavigationRegion3D.
- Add a PlaneMesh to the MeshInstance3D.

- Hit the Editor bake button and a navigation mesh should appear.


Bake NavigationMesh
Clear NavigationMesh
Filter: name, t:type,
Node3D
NavigationRegion3D
MeshInstance3D
Perspective

Baking a navigation mesh with the NavigationServer

The NavigationServer2D and NavigationServer3D have API functions to call each step of the navigation mesh baking process individually.

- `parse_source_geometry_data()` can be used to parse source geometry to a reusable and serializable resource.
- `bake_from_source_geometry_data()` can be used to bake a navigation mesh from already parsed data e.g. to avoid runtime performance issues with (redundant) parsing.
- `bake_from_source_geometry_data_async()` is the same but bakes the navigation mesh deferred with threads, not blocking the main thread.

Compared to a NavigationRegion, the NavigationServer offers finer control over the navigation mesh baking process. In turn it is more complex to use but also provides more advanced options.

Some other advantages of the NavigationServer over a NavigationRegion are:

- The server can parse source geometry without baking, e.g. to cache it for later use.
- The server allows selecting the root node at which to start the source geometry parsing manually.
- The server can accept and bake from procedurally generated source geometry data.
- The server can bake multiple navigation meshes in sequence while (re)using the same source geometry data.

To bake navigation meshes with the NavigationServer, source geometry is required. Source geometry is geometry data that should be considered in a navigation mesh baking process. Both navigation meshes for 2D and 3D are created by baking them from source geometry.

2D and 3D versions of the source geometry resources are available as NavigationMeshSourceGeometryData2D and NavigationMeshSourceGeometryData3D respectively.

Source geometry can be geometry parsed from visual meshes, from physics collision, or procedural created arrays of data, like outlines (2D) and triangle faces (3D). For convenience, source geometry is commonly parsed directly from node setups in the SceneTree. For runtime navigation mesh (re)bakes, be aware that the geometry parsing always happens on the main thread.

Note

The SceneTree is not thread-safe. Parsing source geometry from the SceneTree can only be done on the main thread.

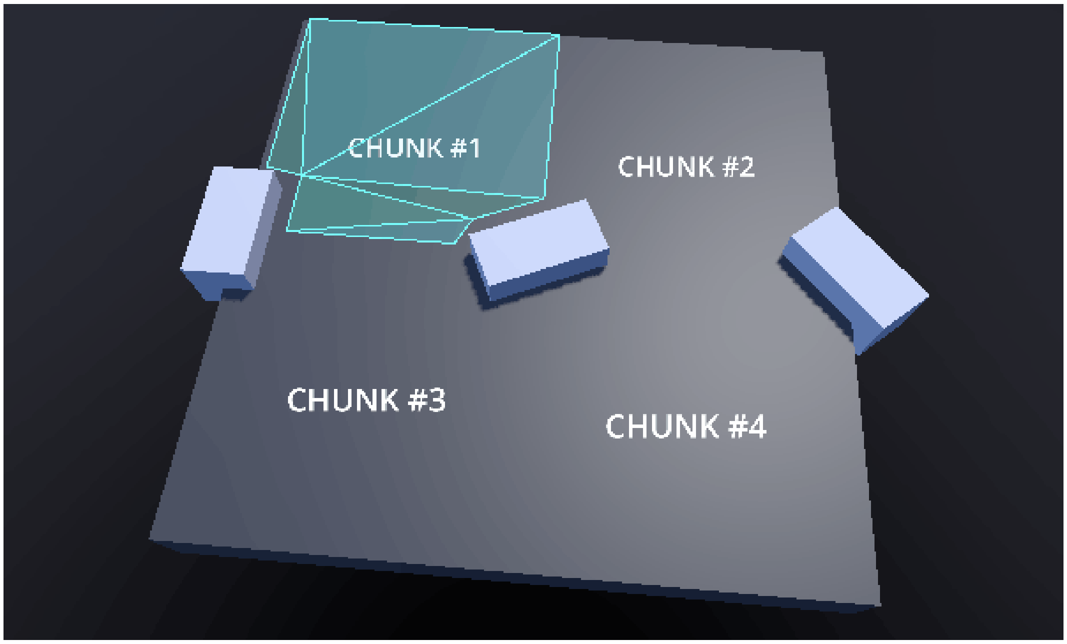
Warning

The data from visual meshes and polygons needs to be received from the GPU, stalling the RenderingServer in the process. For runtime (re)baking prefer using physics shapes as parsed source geometry.

Source geometry is stored inside resources so the created geometry can be reused for multiple bakes. E.g. baking multiple navigation meshes for different agent sizes from the same source geometry. This also allows to save source geometry to disk so it can be loaded later, e.g. to avoid the overhead of parsing it again at runtime.

The geometry data should be in general kept very simple. As many edges as are required but as few as possible. Especially in 2D duplicated and nested geometry should be avoided as it forces polygon hole calculation that can result in flipped polygons. An example for nested geometry would be a smaller StaticBody2D shape placed completely inside the bounds of another StaticBody2D shape.

Baking navigation mesh chunks for large worlds

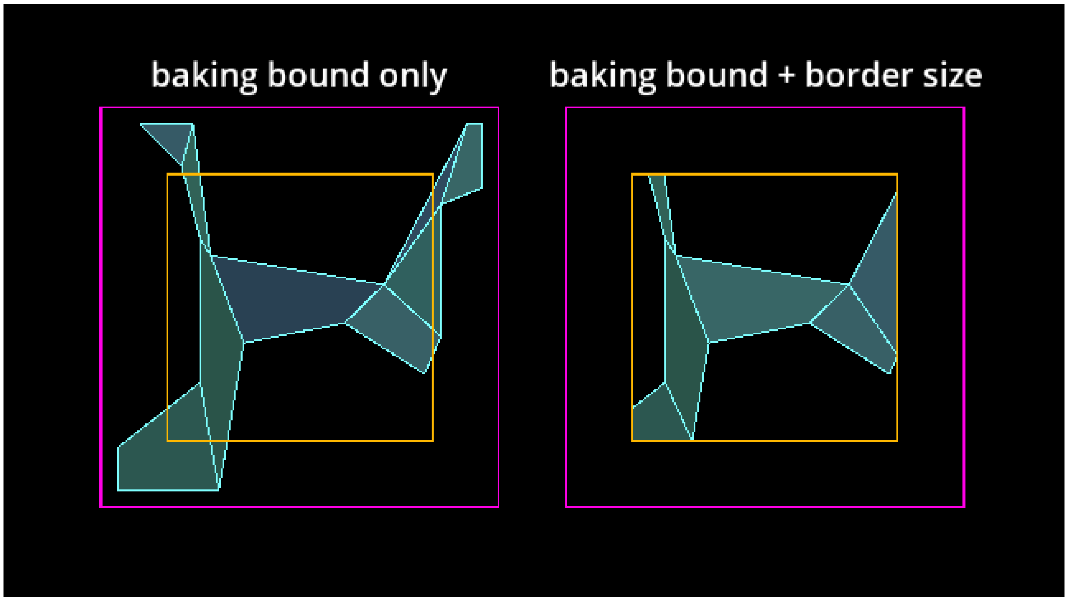


Building and updating individual navigation mesh chunks at runtime.

See also

You can see the navigation mesh chunk baking in action in the Navigation Mesh Chunks 2D [https://github.com/godotengine/godot-demo-projects/tree/master/2d/navigation_mesh_chunks] and Navigation Mesh Chunks 3D [https://github.com/godotengine/godot-demo-projects/tree/master/3d/navigation_mesh_chunks] demo projects.

To avoid misaligned edges between different region chunks the navigation meshes have two important properties for the navigation mesh baking process. The baking bound and the border size. Together they can be used to ensure perfectly aligned edges between region chunks.



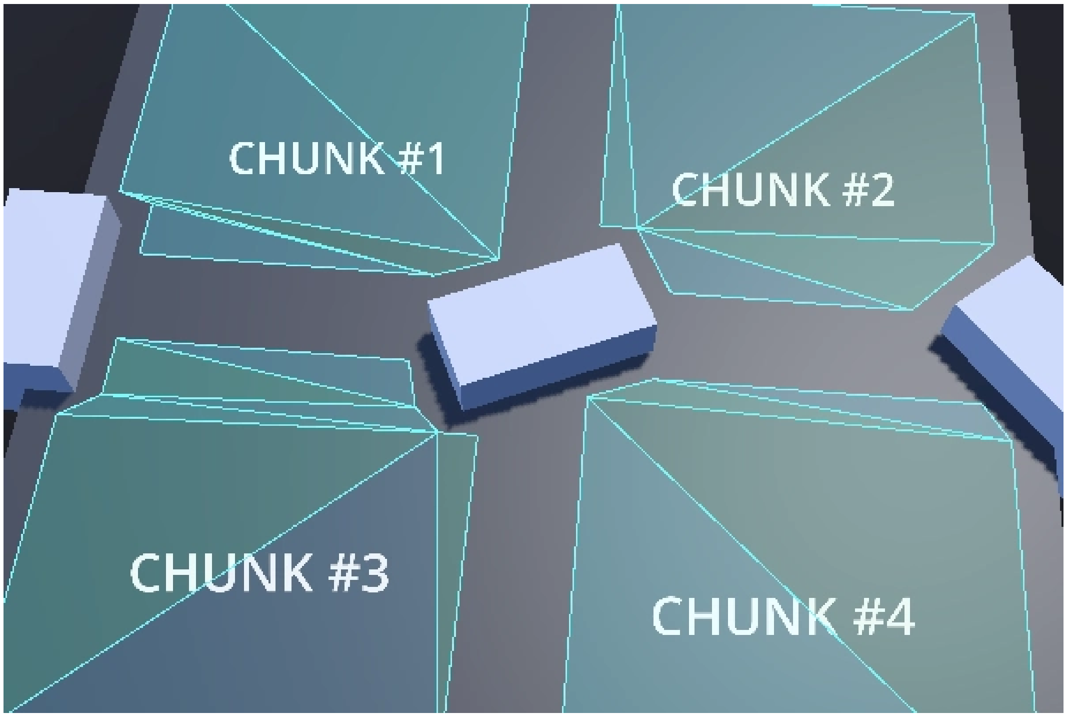
Navigation mesh chunk baked with bake bound or baked with additional border size.

The baking bound, which is an axis-aligned Rect2 for 2D and AABB for 3D, limits the used source geometry by discarding all the geometry that is outside of the bounds.

The NavigationPolygon properties `baking_rect` and `baking_rect_offset` can be used to create and place the 2D baking bound.

The NavigationMesh properties `filter_baking_aabb` and `filter_baking_aabb_offset` can be used to create and place the 3D baking bound.

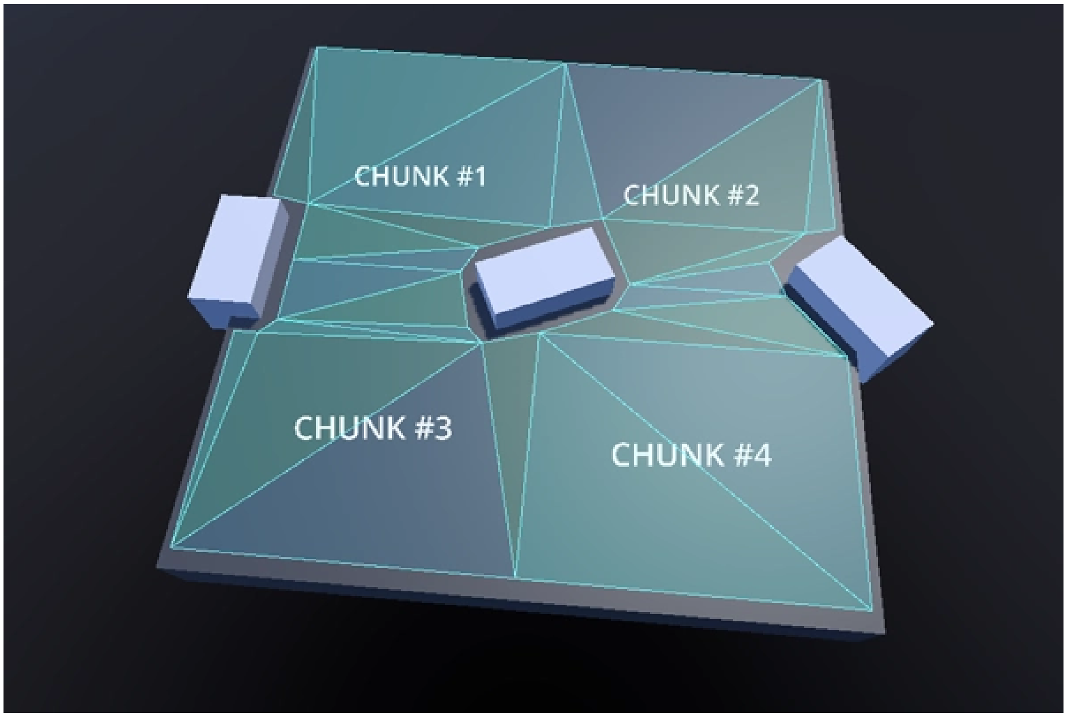
With only the baking bound set another problem still exists. The resulting navigation mesh will inevitably be affected by necessary offsets like the `agent_radius` which makes the edges not align properly.



Navigation mesh chunks with noticeable gaps due to baked agent radius offset.

This is where the `border_size` property for navigation mesh comes in. The border size is an inward margin from the baking bound. The important characteristic of the border size is that it is unaffected by most offsets and postprocessing like the `agent_radius`.

Instead of discarding source geometry, the border size discards parts of the final surface of the baked navigation mesh. If the baking bound is large enough the border size can remove the problematic surface parts so that only the intended chunk size is left.



Navigation mesh chunks with aligned edges and without gaps.

Note

The baking bounds need to be large enough to include a reasonable amount of source geometry from all the neighboring chunks.

Warning

In 3D the functionality of the border size is limited to the xz-axis.

Navigation mesh baking common problems

There are some common user problems and important caveats to consider when creating or baking navigation meshes.

- Navigation mesh baking creates frame rate problems at runtime

  The navigation mesh baking is by default done on a background thread, so as long as the platform supports threads, the actual baking is rarely the source of any performance issues (assuming a reasonably sized and complex geometry for runtime rebakes).

  The common source for performance issues at runtime is the parsing step for source geometry that involves nodes and the SceneTree. The SceneTree is not thread-safe so all the nodes need to be parsed on the main thread. Some nodes with a lot of data can be very heavy and slow to parse at runtime, e.g. a TileMap has one or more polygons for every single used cell and TileMapLayer to parse. Nodes that hold meshes need to request the data from the RenderingServer stalling the rendering in the process.

  To improve performance, use more optimized shapes, e.g. collision shapes over detailed visual meshes, and merge and simplify as much geometry as possible upfront. If nothing helps, don't parse the SceneTree and add the source geometry procedural with scripts. If only pure data arrays are used as source geometry, the entire baking process can be done on a background thread.

- Navigation mesh creates unintended holes in 2D.

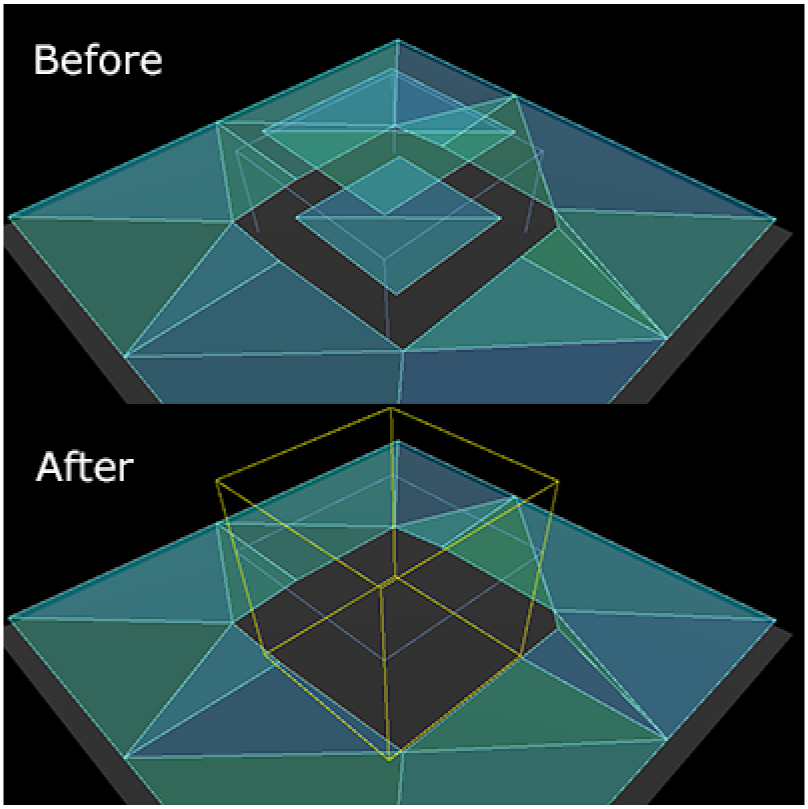
  The navigation mesh baking in 2D is done by doing polygon clipping operations based on outline paths. Polygons with "holes" are a necessary evil to create more complex 2D polygons but can become unpredictable for users with many complex shapes involved.

To avoid any unexpected problems with polygon hole calculations, avoid nesting any outlines inside other outlines of the same type (traversable / obstruction). This includes the parsed shapes from nodes. E.g. placing a smaller StaticBody2D shape inside a larger StaticBody2D shape can result in the resulting polygon being flipped.

- Navigation mesh appears inside geometry in 3D.

  The navigation mesh baking in 3D has no concept of "inside". The voxel cells used to rasterize the geometry are either occupied or not. Remove the geometry that is on the ground inside the other geometry. If that is not possible, add smaller "dummy" geometry inside with as few triangles as possible so the cells are occupied with something.

  A NavigationObstacle3D shape set to bake with navigation mesh can be used to discard geometry as well.



A NavigationObstacle3D shape can be used to discard unwanted navigation mesh parts.

Navigation mesh script templates

The following script uses the NavigationServer to parse source geometry from the scene tree, bakes a navigation mesh, and updates a navigation region with the updated navigation mesh.

2D GDScript

```
extends Node2D

var navigation_mesh: NavigationPolygon
var source_geometry : NavigationMeshSourceGeometryData2D
var callback_parsing : Callable
var callback_baking : Callable
var region_rid: RID

func _ready() -> void:
    navigation_mesh = NavigationPolygon.new()
    navigation_mesh.agent_radius = 10.0
    source_geometry = NavigationMeshSourceGeometryData2D.new()
    callback_parsing = on_parsing_done
    callback_baking = on_baking_done
    region_rid = NavigationServer2D.region_create()

    # Enable the region and set it to the default navigation map.
    NavigationServer2D.region_set_enabled(region_rid, true)
    NavigationServer2D.region_set_map(region_rid, get_world_2d().get_navigation_map())

    # Some mega-nodes like TileMap are often not ready on the first frame.
    # Also the parsing needs to happen on the main-thread.
    # So do a deferred call to avoid common parsing issues.
    parse_source_geometry.call_deferred()

func parse_source_geometry() -> void:
    source_geometry.clear()
    var root_node: Node2D = self

    # Parse the obstruction outlines from all child nodes of
the root node by default.
    NavigationServer2D.parse_source_geometry_data(
        navigation_mesh,
        source_geometry,
        root_node,
        callback_parsing
    )

func on_parsing_done() -> void:
    # If we did not parse a TileMap with navigation mesh cells
we may now only
    # have obstruction outlines so add at least one traversable
outline
    # so the obstructions outlines have something to "cut"
into.

source_geometry.add_traversable_outline(PackedVector2Array([
        Vector2(0.0, 0.0),
        Vector2(500.0, 0.0),
        Vector2(500.0, 500.0),
        Vector2(0.0, 500.0)
    ]))

    # Bake the navigation mesh on a thread with the source
geometry data.
    NavigationServer2D.bake_from_source_geometry_data_async(
        navigation_mesh,
        source_geometry,
        callback_baking
    )

func on_baking_done() -> void:
    # Update the region with the updated navigation mesh.

NavigationServer2D.region_set_navigation_polygon(region_rid,
navigation_mesh)
```

3D GDScript

```
extends Node3D

var navigation_mesh: NavigationMesh
var source_geometry : NavigationMeshSourceGeometryData3D
var callback_parsing : Callable
var callback_baking : Callable
var region_rid: RID

func _ready() -> void:
    navigation_mesh = NavigationMesh.new()
    navigation_mesh.agent_radius = 0.5
    source_geometry = NavigationMeshSourceGeometryData3D.new()
    callback_parsing = on_parsing_done
    callback_baking = on_baking_done
    region_rid = NavigationServer3D.region_create()

    # Enable the region and set it to the default navigation map.
    NavigationServer3D.region_set_enabled(region_rid, true)
    NavigationServer3D.region_set_map(region_rid, get_world_3d().get_navigation_map())

    # Some mega-nodes like GridMap are often not ready on the first frame.
    # Also the parsing needs to happen on the main-thread.
    # So do a deferred call to avoid common parsing issues.
    parse_source_geometry.call_deferred()

func parse_source_geometry() -> void:
    source_geometry.clear()
    var root_node: Node3D = self

    # Parse the geometry from all mesh child nodes of the root node by default.
    NavigationServer3D.parse_source_geometry_data(
        navigation_mesh,
        source_geometry,
        root_node,
        callback_parsing
    )

func on_parsing_done() -> void:
    # Bake the navigation mesh on a thread with the source
```

```
geometry data.
    NavigationServer3D.bake_from_source_geometry_data_async(
        navigation_mesh,
        source_geometry,
        callback_baking
    )

func on_baking_done() -> void:
    # Update the region with the updated navigation mesh.
    NavigationServer3D.region_set_navigation_mesh(region_rid,
navigation_mesh)
```

The following script uses the NavigationServer to update a navigation region with procedurally generated navigation mesh data.

2D GDScript

```
extends Node2D

var navigation_mesh: NavigationPolygon
var region_rid: RID

func _ready() -> void:
    navigation_mesh = NavigationPolygon.new()
    region_rid = NavigationServer2D.region_create()

    # Enable the region and set it to the default navigation
map.
    NavigationServer2D.region_set_enabled(region_rid, true)
    NavigationServer2D.region_set_map(region_rid,
get_world_2d().get_navigation_map())

    # Add vertices for a convex polygon.
    navigation_mesh.vertices = PackedVector2Array([
        Vector2(0.0, 0.0),
        Vector2(100.0, 0.0),
        Vector2(100.0, 100.0),
        Vector2(0.0, 100.0)
    ])

    # Add indices for the polygon.
    navigation_mesh.add_polygon(
        PackedInt32Array([0, 1, 2, 3])
    )


NavigationServer2D.region_set_navigation_polygon(region_rid,
navigation_mesh)
```

3D GDScript

```
extends Node3D

var navigation_mesh: NavigationMesh
var region_rid: RID

func _ready() -> void:
    navigation_mesh = NavigationMesh.new()
    region_rid = NavigationServer3D.region_create()

    # Enable the region and set it to the default navigation
map.
    NavigationServer3D.region_set_enabled(region_rid, true)
    NavigationServer3D.region_set_map(region_rid,
get_world_3d().get_navigation_map())

    # Add vertices for a convex polygon.
    navigation_mesh.vertices = PackedVector3Array([
        Vector3(-1.0, 0.0, 1.0),
        Vector3(1.0, 0.0, 1.0),
        Vector3(1.0, 0.0, -1.0),
        Vector3(-1.0, 0.0, -1.0),
    ])

    # Add indices for the polygon.
    navigation_mesh.add_polygon(
        PackedInt32Array([0, 1, 2, 3])
    )

    NavigationServer3D.region_set_navigation_mesh(region_rid,
navigation_mesh)
```

Using NavigationPaths

Obtaining a NavigationPath

Navigation paths can be directly queried from the NavigationServer and do not require any additional nodes or objects as long as the navigation map has a navigation mesh to work with.

To obtain a 2D path, use `NavigationServer2D.map_get_path(map, from, to, optimize, navigation_layers)`.

To obtain a 3D path, use `NavigationServer3D.map_get_path(map, from, to, optimize, navigation_layers)`.

For more customizable navigation path queries that require additional setup see Using NavigationPathQueryObjects.

One of the required parameters for the query is the RID of the navigation map. Each game world has a default navigation map automatically created. The default navigation maps can be retrieved with `get_world_2d().get_navigation_map()` from any Node2D inheriting node or `get_world_3d().get_navigation_map()` from any Node3D inheriting node. The second and third parameters are the starting position and the target position as Vector2 for 2D or Vector3 for 3D.

If the `optimized` parameter is `true`, path positions will be shortened along polygon corners with an additional funnel algorithm pass. This works well for free movement on navigation meshes with unequally sized polygons as the path will hug around corners along the polygon corridor found by the A* algorithm. With small cells the A* algorithm creates a very narrow funnel corridor that can create ugly corner paths when used with grids.

If the `optimized` parameter is `false`, path positions will be placed at the center of each polygon edge. This works well for pure grid movement on navigation meshes with equally sized polygons as the path will go through the center of the grid cells. Outside of grids due to polygons often covering large open areas with a single, long edge this can create paths with unnecessary long detours.

2D GDScript

```
extends Node2D

# Basic query for a navigation path using the default
navigation map.

func get_navigation_path(p_start_position: Vector2,
p_target_position: Vector2) -> PackedVector2Array:
    if not is_inside_tree():
        return PackedVector2Array()

    var default_map_rid: RID =
get_world_2d().get_navigation_map()
    var path: PackedVector2Array =
NavigationServer2D.map_get_path(
        default_map_rid,
        p_start_position,
        p_target_position,
        true
    )
    return path
```

3D GDScript

```
extends Node3D

# Basic query for a navigation path using the default
navigation map.

func get_navigation_path(p_start_position: Vector3,
p_target_position: Vector3) -> PackedVector3Array:
    if not is_inside_tree():
        return PackedVector3Array()
    var default_map_rid: RID =
get_world_3d().get_navigation_map()
    var path: PackedVector3Array =
NavigationServer3D.map_get_path(
        default_map_rid,
        p_start_position,
        p_target_position,
        true
    )
    return path
```

A returned `path` by the NavigationServer will be a `PackedVector2Array` for 2D or a `PackedVector3Array` for 3D. These are just a memory-optimized `Array` of vector positions. All position vectors inside the array are guaranteed to be inside a NavigationPolygon or NavigationMesh. The path array, if not empty, has the navigation mesh position closest to the starting position at the first index `path[0]` position. The closest available navigation mesh position to the target position is the last index `path[path.size()-1]` position. All indexes between are the path points that an actor should follow to reach the target without leaving the navigation mesh.

Note

If the target position is on a different navigation mesh that is not merged or connected the navigation path will lead to the closest possible position on the starting position navigation mesh.

The following script moves a Node3D inheriting node along a navigation path using the default navigation map by setting the target position with `set_movement_target()`.

GDScript

```
@onready var default_3d_map_rid: RID =
get_world_3d().get_navigation_map()

var movement_speed: float = 4.0
var movement_delta: float
var path_point_margin: float = 0.5

var current_path_index: int = 0
var current_path_point: Vector3
var current_path: PackedVector3Array

func set_movement_target(target_position: Vector3):

    var start_position: Vector3 = global_transform.origin

    current_path = NavigationServer3D.map_get_path(
        default_3d_map_rid,
        start_position,
        target_position,
        true
    )

    if not current_path.is_empty():
        current_path_index = 0
        current_path_point = current_path[0]

func _physics_process(delta):

    if current_path.is_empty():
        return

    movement_delta = movement_speed * delta

    if global_transform.origin.distance_to(current_path_point)
<= path_point_margin:
        current_path_index += 1
        if current_path_index >= current_path.size():
            current_path = []
            current_path_index = 0
            current_path_point = global_transform.origin
            return

    current_path_point = current_path[current_path_index]
    var new_velocity: Vector3 =
global_transform.origin.direction_to(current_path_point) *
movement_delta

    global_transform.origin =
global_transform.origin.move_toward(global_transform.origin +
new_velocity, movement_delta)
```

Using NavigationPathQueryObjects

`NavigationPathQueryObjects` can be used together with `NavigationServer.query_path()` to obtain a heavily **customized** navigation path including optional **meta data** about the path.

This requires more setup compared to obtaining a normal NavigationPath but lets you tailor the pathfinding and provided path data to the different needs of a project.

NavigationPathQueryObjects consist of a pair of objects, a `NavigationPathQueryParameters` object holding the customization options for the query and a `NavigationPathQueryResult` that receives (regular) updates with the resulting path and meta data from the query.

2D and 3D versions of `NavigationPathQueryParameters` are available as NavigationPathQueryParameters2D and NavigationPathQueryParameters3D respectively.

2D and 3D versions of `NavigationPathQueryResult` are available as NavigationPathQueryResult2D and NavigationPathQueryResult3D respectively.

Both parameters and result are used as a pair with the `NavigationServer.query_path()` function.

For the available customization options and their use see the class doc of the parameters.

While not a strict requirement, both objects are intended to be created once in advance, stored in a persistent variable for the agent and reused for every followup path query with updated parameters. This reuse avoids performance implications from frequent object creation if a project has a large quantity of simultaneous agents that regularly update their paths.

2D GDScript

```
# Prepare query objects.
var query_parameters := NavigationPathQueryParameters2D.new()
var query_result := NavigationPathQueryResult2D.new()

func query_path(p_start_position: Vector2, p_target_position:
Vector2, p_navigation_layers: int = 1) -> PackedVector2Array:
    if not is_inside_tree():
        return PackedVector2Array()

    query_parameters.map = get_world_2d().get_navigation_map()
    query_parameters.start_position = p_start_position
    query_parameters.target_position = p_target_position
    query_parameters.navigation_layers = p_navigation_layers

    NavigationServer2D.query_path(query_parameters,
query_result)
    var path: PackedVector2Array = query_result.get_path()

    return path
```

3D GDScript

```
# Prepare query objects.
var query_parameters := NavigationPathQueryParameters3D.new()
var query_result := NavigationPathQueryResult3D.new()

func query_path(p_start_position: Vector3, p_target_position:
Vector3, p_navigation_layers: int = 1) -> PackedVector3Array:
    if not is_inside_tree():
        return PackedVector3Array()

    query_parameters.map = get_world_3d().get_navigation_map()
    query_parameters.start_position = p_start_position
    query_parameters.target_position = p_target_position
    query_parameters.navigation_layers = p_navigation_layers
    NavigationServer3D.query_path(query_parameters, query_result)
    var path: PackedVector3Array = query_result.get_path()

    return path
```

Using NavigationAgents

NavigationsAgents are helper nodes that combine functionality for pathfinding, path following and agent avoidance for a Node2D/3D inheriting parent node. They facilitate common calls to the NavigationServer API on behalf of the parent actor node in a more convenient manner for beginners.

2D and 3D version of NavigationAgents are available as NavigationAgent2D and NavigationAgent3D respectively.

New NavigationAgent nodes will automatically join the default navigation map on the World2D/World3D.

NavigationsAgent nodes are optional and not a hard requirement to use the navigation system. Their entire functionality can be replaced with scripts and direct calls to the NavigationServer API.

NavigationAgent Pathfinding

NavigationAgents query a new navigation path on their current navigation map when their `target_position` is set with a global position.

The result of the pathfinding can be influenced with the following properties.

- The `navigation_layers` bitmask can be used to limit the navigation meshes that the agent can use.
- The `pathfinding_algorithm` controls how the pathfinding travels through the navigation mesh polygons in the path search.
- The `path_postprocessing` sets if or how the raw path corridor found by the pathfinding is altered before it is returned.

- The `path_metadata_flags` enable the collection of additional path point meta data returned by the path.
- The `simplify_path` and `simplify_epsilon` properties can be used to remove less critical points from the path.

Warning

Disabling path meta flags will disable related signal emissions on the agent.

NavigationAgent Pathfollowing

After a `target_position` has been set for the agent, the next position to follow in the path can be retrieved with the `get_next_path_position()` function.

Once the next path position is received move the parent actor node of the agent towards this path position with your own movement code.

Note

The navigation system never moves the parent node of a NavigationAgent. The movement is entirely in the hands of users and their custom scripts.

NavigationAgents have their own internal logic to proceed with the current path and call for updates.

The `get_next_path_position()` function is responsible for updating many of the agent's internal states and properties. The function should be repeatedly called *once* every `physics_process` until `is_navigation_finished()` tells that the path is finished. The function should not be called after the target position or path end has been reached as it can make the agent jitter in place due to the repeated path updates. Always check very early in script with `is_navigation_finished()` if the path is already finished.

The following properties influence the path following behavior.

- The `path_desired_distance` defines the distance at which the agent advances its internal path index to the next path position.
- The `target_desired_distance` defines the distance at which the agent considers the target position to be reached and the path at its end.
- The `path_max_distance` defines when an agent requests a new path cause it was moved too far away from the current path point segment.

The important updates are all triggered with the `get_next_path_position()` function when called in `_physics_process()`.

NavigationAgents can be used with `process` but are still limited to a single update that happens in `physics_process`.

Script examples for various nodes commonly used with NavigationAgents can be found further below.

Pathfollowing common problems

There are some common user problems and important caveats to consider when writing agent movement scripts.

- The path is returned empty

  If an agent queries a path before the navigation map synchronisation, e.g. in a `_ready()` function, the path might return empty. In this case the `get_next_path_position()` function will return the same position as the agent parent node and the agent will consider the path end reached. This is fixed by making a deferred call or using a callback e.g. waiting for the navigation map changed signal.

- The agent is stuck dancing between two positions

  This is usually caused by very frequent path updates every single frame, either deliberate or by accident (e.g. max path distance set too short). The pathfinding needs to find the closest position that are valid on navigation mesh. If a new path is requested every single frame the first path positions might end up switching constantly in front and behind the agent's current position, causing it to dance between the two positions.

- The agent is backtracking sometimes

  If an agent moves very fast it might overshoot the path_desired_distance check without ever advancing the path index. This can lead to the agent backtracking to the path point now behind it until it passes the distance check to increase the path index. Increase the desired distances accordingly for your agent speed and update rate usually fixes this as well as a more balanced navigation mesh polygon layout with not too many polygon edges cramped together in small spaces.

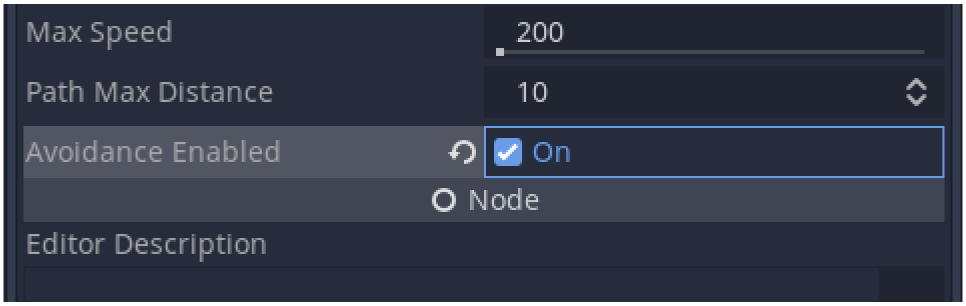
- The agent is sometimes looking backwards for a frame

  Same as with stuck dancing agents between two positions, this is usually caused by very frequent path updates every single frame. Depending on your navigation mesh layout, and especially when an agent is directly placed over a navigation mesh edge or edge connection, expect path positions to be sometimes slightly "behind" your actors current orientation. This happens due to precision issues and can not always be avoided. This is usually only a visible problem if actors are instantly rotated to face the current path position.

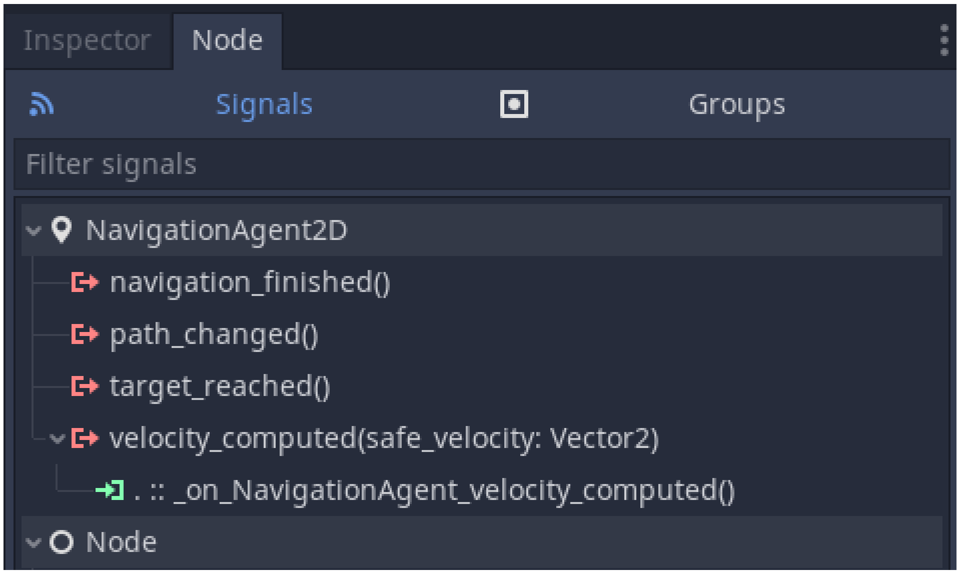
NavigationAgent Avoidance

This section explains how to use the navigation avoidance specific to NavigationAgents.

In order for NavigationAgents to use the avoidance feature the `enable_avoidance` property must be set to `true`.


Max Speed 200
Path Max Distance 10
Avoidance Enabled ☑ On

The `velocity_computed` signal of the NavigationAgent node must be connected to receive the safe velocity calculation result.



NavigationAgent2D
navigation_finished()
path_changed()
target_reached()
velocity_computed(safe_velocity: Vector2)
. :: _on_NavigationAgent_velocity_computed()
Node

Set the `velocity` of the NavigationAgent node in `_physics_process()` to update the agent with the current velocity of the agent's parent node.

While avoidance is enabled on the agent the `safe_velocity` vector will be received with the velocity_computed signal every physics frame. This velocity vector should be used to move the NavigationAgent's parent node in order to avoidance collision with other avoidance using agents or avoidance obstacles.

Note

Only other agents on the same map that are registered for avoidance themself will be considered in the avoidance calculation.

The following NavigationAgent properties are relevant for avoidance:

- The property `height` is available in 3D only. The height together with the current global y-axis position of the agent determines the vertical placement of the agent in the avoidance simulation. Agents using the 2D avoidance will automatically ignore other agents or obstacles that are below or above them.
- The property `radius` controls the size of the avoidance circle, or in case of 3D sphere, around the agent. This area describes the agents body and not the avoidance maneuver distance.
- The property `neighbor_distance` controls the search radius of the agent when searching for other agents that should be avoided. A lower value reduces processing cost.
- The property `max_neighbors` controls how many other agents are considered in the avoidance calculation if they all have overlapping radius. A lower value reduces processing cost but a too low value may result in agents ignoring the avoidance.
- The properties `time_horizon_agents` and `time_horizon_obstacles` control the avoidance prediction time for other agents or obstacles in seconds. When agents

calculate their safe velocities they choose velocities that can be kept for this amount of seconds without colliding with another avoidance object. The prediction time should be kept as low as possible as agents will slow down their velocities to avoid collision in that timeframe.
- The property `max_speed` controls the maximum velocity allowed for the agents avoidance calculation. If the agents parents moves faster than this value the avoidance `safe_velocity` might not be accurate enough to avoid collision.
- The property `use_3d_avoidance` switches the agent between the 2D avoidance (xz axis) and the 3D avoidance (xyz axis) on the next update. Note that 2D avoidance and 3D avoidance run in separate avoidance simulations so agents split between them do not affect each other.
- The properties `avoidance_layers` and `avoidance_mask` are bitmasks similar to e.g. physics layers. Agents will only avoid other avoidance objects that are on an avoidance layer that matches at least one of their own avoidance mask bits.
- The `avoidance_priority` makes agents with a higher priority ignore agents with a lower priority. This can be used to give certain agents more importance in the avoidance simulation, e.g. important npcs characters, without constantly changing their entire avoidance layers or mask.

Avoidance exists in its own space and has no information from navigation meshes or physics collision. Behind the scene avoidance agents are just circles with different radius on a flat 2D plane or spheres in an otherwise empty 3D space. NavigationObstacles can be used to add some environment constrains to the avoidance simulation, see Using NavigationObstacles.

Note

Avoidance does not affect the pathfinding. It should be seen as an additional option for constantly moving objects that cannot be (re)baked to a navigation mesh efficiently in order to move around them.

Note

RVO avoidance makes implicit assumptions about natural agent behavior. E.g. that agents move on reasonable passing sides that can be assigned when they encounter each other. This means that very clinical avoidance test scenarios will commonly fail. E.g. agents moved directly against each other with perfect opposite velocities will fail because the agents can not get their passing sides assigned.

Using the NavigationAgent `enable_avoidance` property is the preferred option to toggle avoidance. The following code snippets can be used to toggle avoidance on agents, create or delete avoidance callbacks or switch avoidance modes.

2D GDScript

```
extends NavigationAgent2D

func _ready() -> void:
    var agent: RID = get_rid()
    # Enable avoidance
    NavigationServer2D.agent_set_avoidance_enabled(agent, true)
    # Create avoidance callback
    NavigationServer2D.agent_set_avoidance_callback(agent,
Callable(self, "_avoidance_done"))

    # Disable avoidance
    NavigationServer2D.agent_set_avoidance_enabled(agent,
false)
    # Delete avoidance callback
    NavigationServer2D.agent_set_avoidance_callback(agent,
Callable())
```

3D GDScript

```
extends NavigationAgent3D

func _ready() -> void:
    var agent: RID = get_rid()
    # Enable avoidance
    NavigationServer3D.agent_set_avoidance_enabled(agent, true)
    # Create avoidance callback
    NavigationServer3D.agent_set_avoidance_callback(agent,
Callable(self, "_avoidance_done"))
    # Switch to 3D avoidance
    NavigationServer3D.agent_set_use_3d_avoidance(agent, true)

    # Disable avoidance
    NavigationServer3D.agent_set_avoidance_enabled(agent,
false)
    # Delete avoidance callback
    NavigationServer3D.agent_set_avoidance_callback(agent,
Callable())
    # Switch to 2D avoidance
    NavigationServer3D.agent_set_use_3d_avoidance(agent, false)
```

NavigationAgent Script Templates

The following sections provides script templates for nodes commonly used with NavigationAgents.

2D GDScript

Node2D

```
extends Node2D

@export var movement_speed: float = 4.0
@onready var navigation_agent: NavigationAgent2D =
get_node("NavigationAgent2D")
var movement_delta: float

func _ready() -> void:

navigation_agent.velocity_computed.connect(Callable(_on_velocit
y_computed))

func set_movement_target(movement_target: Vector2):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is
empty.
    if
NavigationServer2D.map_get_iteration_id(navigation_agent.get_na
vigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    movement_delta = movement_speed * delta
    var next_path_position: Vector2 =
navigation_agent.get_next_path_position()
    var new_velocity: Vector2 =
global_position.direction_to(next_path_position) *
movement_delta
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector2) -> void:
    global_position =
global_position.move_toward(global_position + safe_velocity,
movement_delta)
```

CharacterBody2D

```
extends CharacterBody2D

@export var movement_speed: float = 4.0
```

```
@onready var navigation_agent: NavigationAgent2D =
get_node("NavigationAgent2D")

func _ready() -> void:

navigation_agent.velocity_computed.connect(Callable(_on_velocit
y_computed))

func set_movement_target(movement_target: Vector2):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is
empty.
    if
NavigationServer2D.map_get_iteration_id(navigation_agent.get_na
vigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    var next_path_position: Vector2 =
navigation_agent.get_next_path_position()
    var new_velocity: Vector2 =
global_position.direction_to(next_path_position) *
movement_speed
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector2):
    velocity = safe_velocity
    move_and_slide()
```

RigidBody2D

```
extends RigidBody2D

@export var movement_speed: float = 4.0
@onready var navigation_agent: NavigationAgent2D =
get_node("NavigationAgent2D")
```

```
func _ready() -> void:
navigation_agent.velocity_computed.connect(Callable(_on_velocity_computed))

func set_movement_target(movement_target: Vector2):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is empty.
    if NavigationServer2D.map_get_iteration_id(navigation_agent.get_navigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    var next_path_position: Vector2 = navigation_agent.get_next_path_position()
    var new_velocity: Vector2 = global_position.direction_to(next_path_position) * movement_speed
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector2):
    linear_velocity = safe_velocity
```

3D GDScript

Node3D

```
extends Node3D

@export var movement_speed: float = 4.0
@onready var navigation_agent: NavigationAgent3D = get_node("NavigationAgent3D")
```

```
var movement_delta: float

func _ready() -> void:

navigation_agent.velocity_computed.connect(Callable(_on_velocit
y_computed))

func set_movement_target(movement_target: Vector3):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is
empty.
    if
NavigationServer3D.map_get_iteration_id(navigation_agent.get_na
vigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    movement_delta = movement_speed * delta
    var next_path_position: Vector3 =
navigation_agent.get_next_path_position()
    var new_velocity: Vector3 =
global_position.direction_to(next_path_position) *
movement_delta
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector3) -> void:
    global_position =
global_position.move_toward(global_position + safe_velocity,
movement_delta)
```

CharacterBody3D

```
extends CharacterBody3D

@export var movement_speed: float = 4.0
@onready var navigation_agent: NavigationAgent3D =
get_node("NavigationAgent3D")

func _ready() -> void:

navigation_agent.velocity_computed.connect(Callable(_on_velocit
y_computed))

func set_movement_target(movement_target: Vector3):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is
empty.
    if
NavigationServer3D.map_get_iteration_id(navigation_agent.get_na
vigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    var next_path_position: Vector3 =
navigation_agent.get_next_path_position()
    var new_velocity: Vector3 =
global_position.direction_to(next_path_position) *
movement_speed
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector3):
    velocity = safe_velocity
    move_and_slide()
```

RigidBody3D

```
extends RigidBody3D

@export var movement_speed: float = 4.0
@onready var navigation_agent: NavigationAgent3D =
get_node("NavigationAgent3D")
```

```
func _ready() -> void:

navigation_agent.velocity_computed.connect(Callable(_on_velocit
y_computed))

func set_movement_target(movement_target: Vector3):
    navigation_agent.set_target_position(movement_target)

func _physics_process(delta):
    # Do not query when the map has never synchronized and is
empty.
    if
NavigationServer3D.map_get_iteration_id(navigation_agent.get_na
vigation_map()) == 0:
        return
    if navigation_agent.is_navigation_finished():
        return

    var next_path_position: Vector3 =
navigation_agent.get_next_path_position()
    var new_velocity: Vector3 =
global_position.direction_to(next_path_position) *
movement_speed
    if navigation_agent.avoidance_enabled:
        navigation_agent.set_velocity(new_velocity)
    else:
        _on_velocity_computed(new_velocity)

func _on_velocity_computed(safe_velocity: Vector3):
    linear_velocity = safe_velocity
```

Using NavigationObstacles

2D and 3D versions of NavigationObstacles nodes are available as NavigationObstacle2D and NavigationObstacle3D respectively.

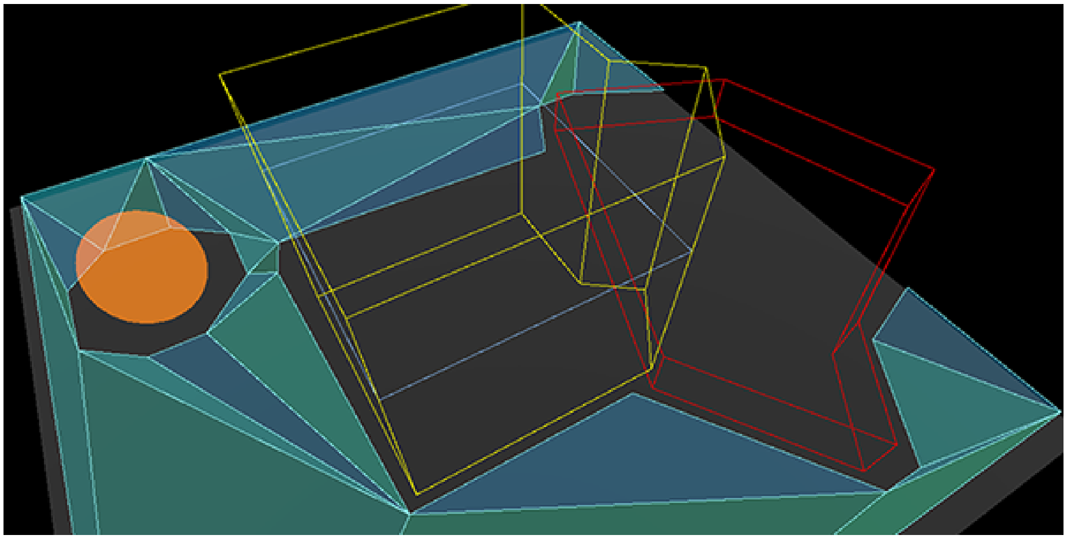
Navigation obstacles are dual purpose in that they can affect both the navigation mesh baking, and the agent avoidance.

- With `affect_navigation_mesh` enabled the obstacle will affect navigation mesh when baked.
- With `avoidance_enabled` the obstacle will affect avoidance agents.

Tip

Avoidance is enabled by default. If the obstacle is not used for avoidance disable `enabled_avoidance` to save performance.

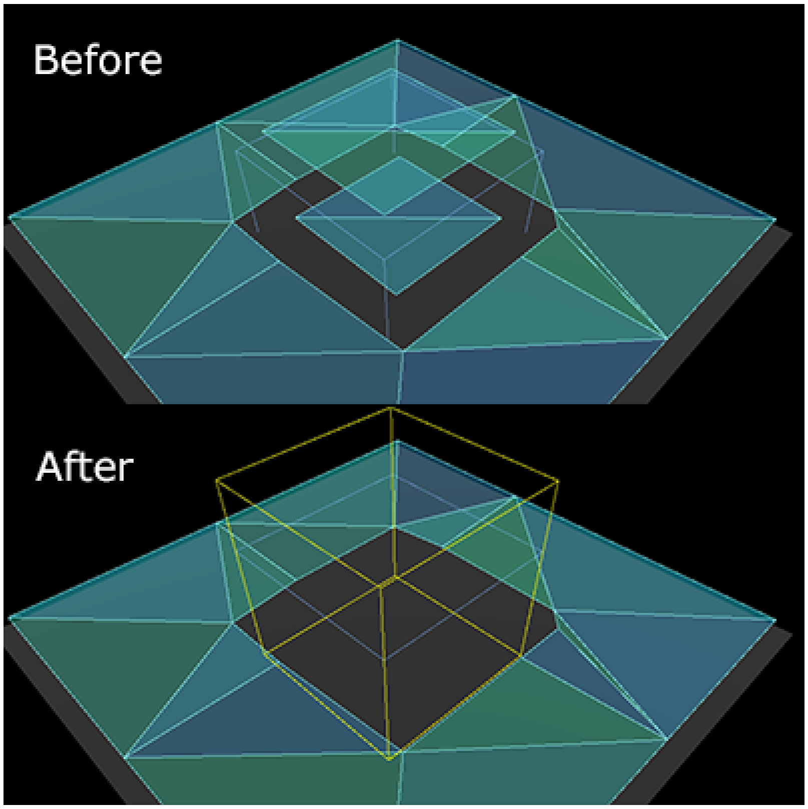
Obstacles and navigation mesh



Navigation obstacles affecting navigation mesh baking.

For navigation mesh baking, obstacles can be used to discard parts of all other source geometry inside the obstacle shape.

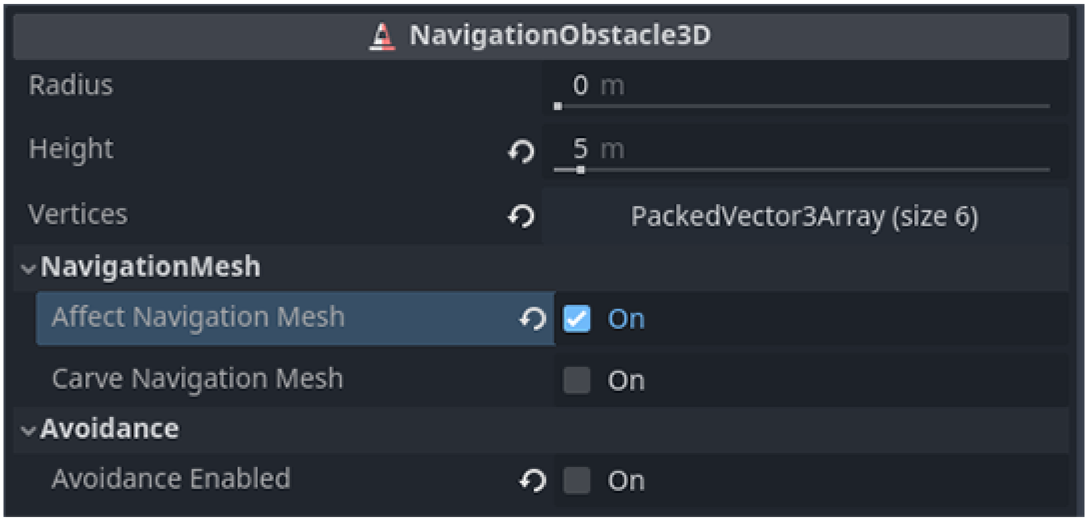
This can be used to stop navigation meshes being baked in unwanted places, e.g. inside "solid" geometry like thick walls or on top of other geometry that should not be included for gameplay like roofs.



Navigation obstacles discard of unwanted navigation mesh.

An obstacle does not add geometry in the baking process, it only removes geometry. It does so by nullifying all the (voxel) cells with rasterized source geometry that are within the obstacle shape. As such its effect and shape detail is limited to the cell resolution used by the baking process.

For more details on the navigation mesh baking see Using navigation meshes.



NavigationObstacle3D
Radius 0 m
Height 5 m
Vertices PackedVector3Array (size 6)
NavigationMesh
Affect Navigation Mesh ☑ On
Carve Navigation Mesh ☐ On
Avoidance
Avoidance Enabled ☐ On

The property `affect_navigation_mesh` makes the obstacle contribute to the navigation mesh baking. It will be parsed or unparsed like all other node objects in a navigation mesh baking process.

The `carve_navigation_mesh` property makes the shape unaffected by offsets of the baking, e.g. the offset added by the navigation mesh `agent_radius`. It will basically act as a stencil and cut into the already offset navigation mesh surface. It will still be affected by further postprocessing of the baking process like edge simplification.

The obstacle shape and placement is defined with the `height` and `vertices` properties, and the `global_position` of the obstacle. The y-axis value of any Vector3 used for the vertices is ignored as the obstacle is projected on a flat horizontal plane.

When baking navigation meshes in scripts obstacles can be added procedurally as a projected obstruction. Obstacles are not involved in the source geometry parsing so adding them just before baking is enough.

2D GDScript

```
var obstacle_outline = PackedVector2Array([
    Vector2(-50, -50),
    Vector2(50, -50),
    Vector2(50, 50),
    Vector2(-50, 50)
])

var navigation_mesh = NavigationPolygon.new()
var source_geometry = NavigationMeshSourceGeometryData2D.new()

NavigationServer2D.parse_source_geometry_data(navigation_mesh,
source_geometry, $MyTestRootNode)

var obstacle_carve: bool = true

source_geometry.add_projected_obstruction(obstacle_outline,
obstacle_carve)

NavigationServer2D.bake_from_source_geometry_data(navigation_mesh, source_geometry)
```

3D GDScript

```
var obstacle_outline = PackedVector3Array([
    Vector3(-5, 0, -5),
    Vector3(5, 0, -5),
    Vector3(5, 0, 5),
    Vector3(-5, 0, 5)
])

var navigation_mesh = NavigationMesh.new()
var source_geometry = NavigationMeshSourceGeometryData3D.new()

NavigationServer3D.parse_source_geometry_data(navigation_mesh,
source_geometry, $MyTestRootNode)

var obstacle_elevation: float =
$MyTestObstacleNode.global_position.y
var obstacle_height: float = 50.0
var obstacle_carve: bool = true

source_geometry.add_projected_obstruction(obstacle_outline,
obstacle_elevation, obstacle_height, obstacle_carve)
```

```
NavigationServer3D.bake_from_source_geometry_data(navigation_mesh, source_geometry)
```

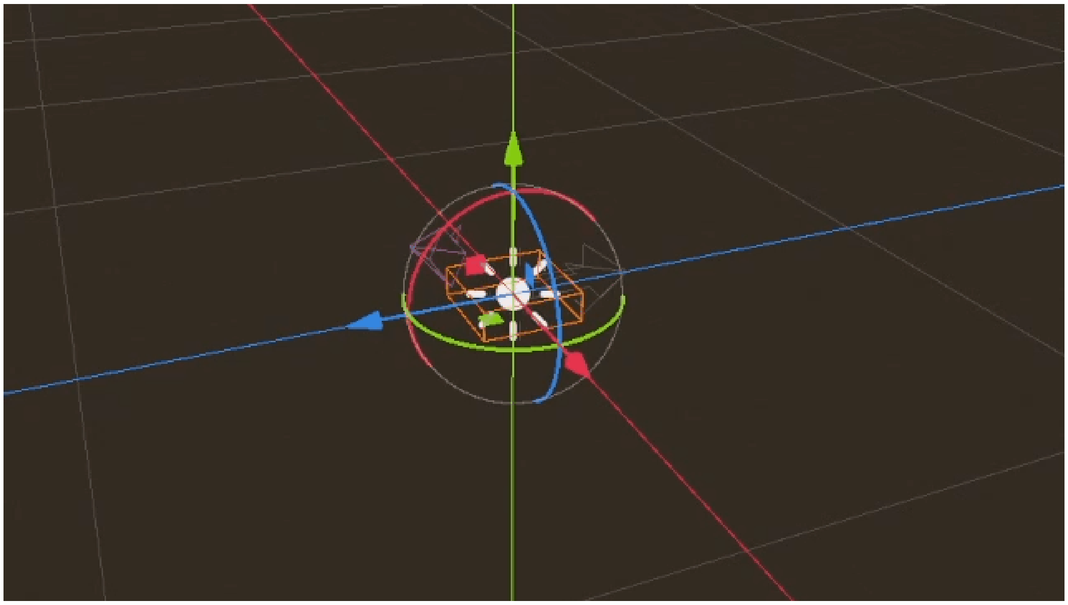
Obstacles and agent avoidance

For avoidance navigation obstacles can be used either as static or dynamic obstacles to affect avoidance controlled agents.

- When used statically NavigationObstacles constrain avoidance controlled agents outside or inside a polygon defined area.
- When used dynamically NavigationObstacles push away avoidance controlled agents in a radius around them.

Static avoidance obstacles

An avoidance obstacle is considered static when its `vertices` property is populated with an outline array of positions to form a polygon.



Static obstacle drawn in the editor to block or contain navigation agents.

- Static obstacles act as hard do-not-cross boundaries for avoidance using agents, e.g. similar to physics collision but for avoidance.
- Static obstacles define their boundaries with an array of outline `vertices` (positions), and in case of 3D with an additional `height` property.
- Static obstacles only work for agents that use the 2D avoidance mode.
- Static obstacles define through winding order of the vertices if agents are pushed out or sucked in.
- Static obstacles can not change their position. They can only be warped to a new position and rebuilt from scratch. Static obstacles as a result are ill-suited for usages where the position is changed every frame, as the constant rebuild has a high performance cost.
- Static obstacles that are warped to another position can not be predicted by agents. This creates the risk of getting agents stuck should a static obstacle be warped on top of agents.

When the 2D avoidance is used in 3D the y-axis of Vector3 vertices is ignored. Instead, the global y-axis position of the obstacle is used as the elevation level. Agents will ignore static obstacles in 3D that are below or above them. This is automatically determined by global y-axis position of both obstacle and agent as the elevation level as well as their respective height properties.

Dynamic avoidance obstacles

An avoidance obstacle is considered dynamic when its `radius` property is greater than zero.

- Dynamic obstacles act as a soft please-move-away-from-me object for avoidance using agents, e.g. similar to how they avoid other agents.
- Dynamic obstacles define their boundaries with a single `radius` for a 2D circle, or in case of 3D avoidance a sphere shape.
- Dynamic obstacles can change their position every frame without additional performance cost.
- Dynamic obstacles with a set velocity can be predicted in their movement by agents.
- Dynamic obstacles are not a reliable way to constrain agents in crowded or narrow spaces.

While both static and dynamic properties can be active at the same time on the same obstacle this is not recommended for performance. Ideally when an obstacle is moving the static vertices are removed and instead the radius activated. When the obstacle reaches the new final position it should gradually enlarge its radius to push all other agents away. With enough created safe space around the obstacle it should add the static vertices again and remove the radius. This helps avoid getting agents stuck in the suddenly appearing static obstacle when the rebuilt static boundary is finished.

Similar to agents the obstacles can make use of the `avoidance_layers` bitmask. All agents with a matching bit on their own avoidance mask will avoid the obstacle.

Procedural obstacles

New obstacles can be created in a script without a Node by using the NavigationServer directly.

Obstacles created with scripts require at least a `map` and a `position`. For dynamic use a `radius` is required. For static use an array of `vertices` is required.

2D GDScript

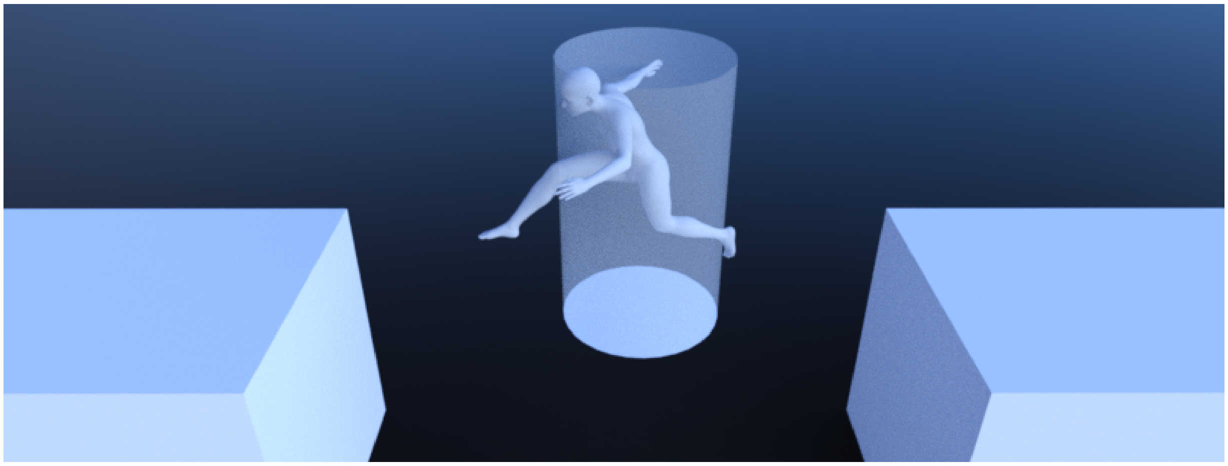
```
# create a new "obstacle" and place it on the default navigation map.
var new_obstacle_rid: RID = NavigationServer2D.obstacle_create()
var default_map_rid: RID = get_world_2d().get_navigation_map()

NavigationServer2D.obstacle_set_map(new_obstacle_rid, default_map_rid)
NavigationServer2D.obstacle_set_position(new_obstacle_rid, global_position)

# Use obstacle dynamic by increasing radius above zero.
NavigationServer2D.obstacle_set_radius(new_obstacle_rid, 5.0)

# Use obstacle static by adding a square that pushes agents out.
var outline = PackedVector2Array([Vector2(-100, -100), Vector2(100, -100), Vector2(100, 100), Vector2(-100, 100)])
NavigationServer2D.obstacle_set_vertices(new_obstacle_rid, outline)

# Enable the obstacle.
NavigationServer2D.obstacle_set_avoidance_enabled(new_obstacle_rid, true)
```

3D GDScript

```
# Create a new "obstacle" and place it on the default navigation map.
var new_obstacle_rid: RID = NavigationServer3D.obstacle_create()
var default_map_rid: RID = get_world_3d().get_navigation_map()

NavigationServer3D.obstacle_set_map(new_obstacle_rid, default_map_rid)
NavigationServer3D.obstacle_set_position(new_obstacle_rid, global_position)

# Use obstacle dynamic by increasing radius above zero.
NavigationServer3D.obstacle_set_radius(new_obstacle_rid, 0.5)
```

```
# Use obstacle static by adding a square that pushes agents out.
var outline = PackedVector3Array([Vector3(-5, 0, -5), Vector3(5, 0, -5), Vector3(5, 0, 5), Vector3(-5, 0, 5)])
NavigationServer3D.obstacle_set_vertices(new_obstacle_rid, outline)
# Set the obstacle height on the y-axis.
NavigationServer3D.obstacle_set_height(new_obstacle_rid, 1.0)

# Enable the obstacle.
NavigationServer3D.obstacle_set_avoidance_enabled(new_obstacle_rid, true)
```

Using NavigationLinks



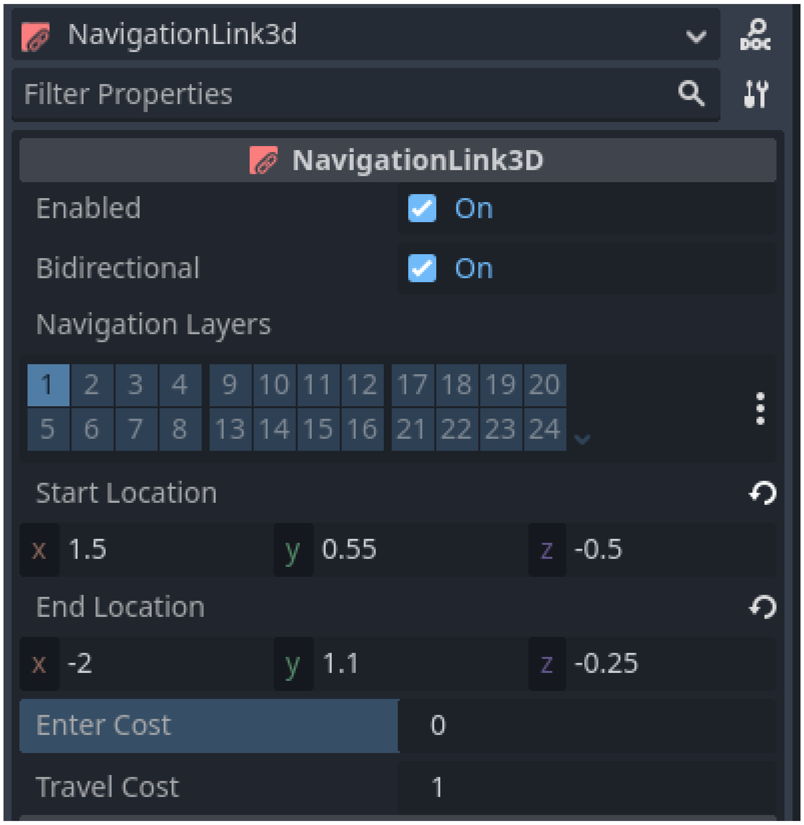
NavigationLinks are used to connect navigation mesh polygons from NavigationRegion2D and NavigationRegion3D over arbitrary distances for pathfinding.

NavigationLinks are also used to consider movement shortcuts in pathfinding available through interacting with gameplay objects e.g. ladders, jump pads or teleports.

2D and 3D versions of NavigationJumplinks nodes are available as NavigationLink2D and NavigationLink3D respectively.

Different NavigationRegions can connect their navigation meshes without the need for a NavigationLink as long as they have overlapping edges or edges that are within navigation map `edge_connection_margin`. As soon as the distance becomes too large, building valid connections becomes a problem - a problem that NavigationLinks can solve.

See Using NavigationRegions to learn more about the use of navigation regions. See Connecting navigation meshes to learn more about how to connect navigation meshes.


NavigationLink3d
Filter Properties
NavigationLink3D
1 2 3 4 9 10 11 12 17 18 19 20
5 6 7 8 13 14 15 16 21 22 23 24

Enabled ☑ On

Bidirectional ☑ On

Navigation Layers

Start Location x 1.5 y 0.55 z -0.5

End Location x -2 y 1.1 z -0.25

Enter Cost 0

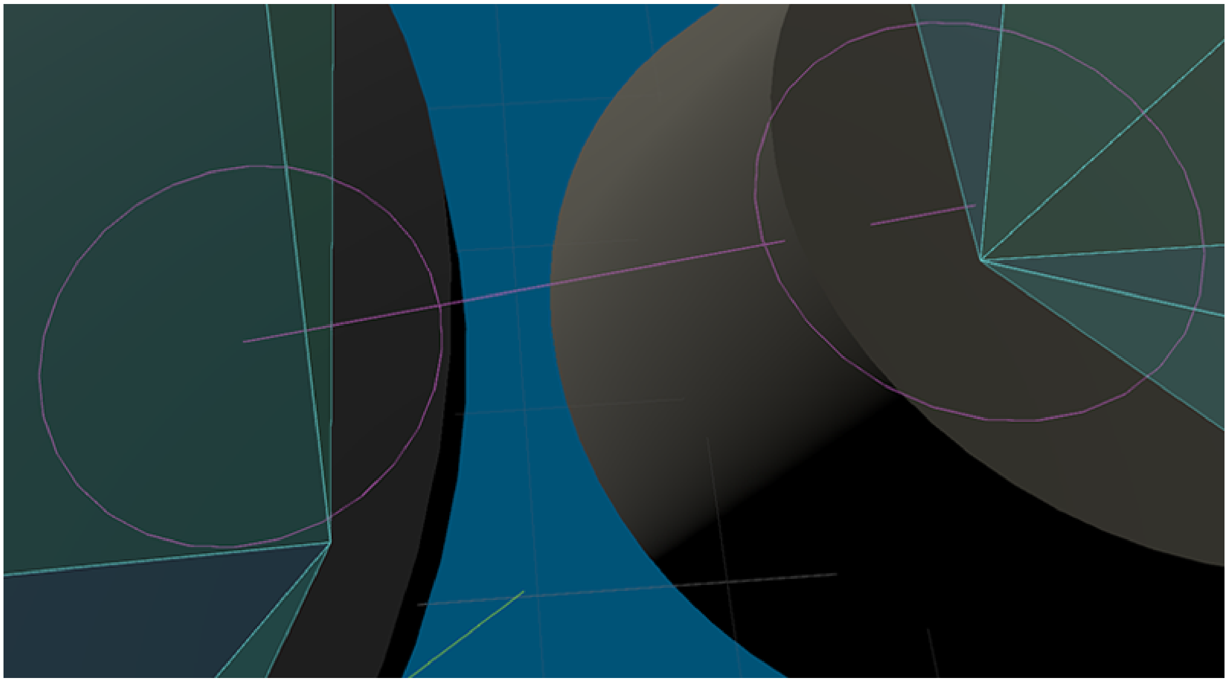
Travel Cost 1

NavigationLinks share many properties with NavigationRegions like `navigation_layers`. NavigationLinks add a single connection between two positions over an arbitrary distance compared to NavigationRegions that add a more local traversable area with a navigation mesh resource.

NavigationLinks have a `start_position` and `end_position` and can go in both directions when `bidirectional` is enabled. When placed a navigationlink connects the navigation mesh polygons closest to its `start_position` and `end_position` within search radius for pathfinding.

The polygon search radius can be configured globally in the ProjectSettings under `navigation/2d_or_3d/default_link_connection_radius` or set for each navigation **map** individually using the `NavigationServer.map_set_link_connection_radius()` function.

Both `start_position` and `end_position` have debug markers in the Editor. The visible radius of a position shows the polygon search radius. All navigation mesh polygons inside are compared and the closest is picked for the edge connection. If no valid polygon is found within the search radius the navigation link gets disabled.



The link debug visuals can be changed in the Editor ProjectSettings under `debug/shapes/navigation`. The visibility of the debug can also be controlled in the Editor 3D Viewport gizmo menu.

A navigation link does not provide any specialized movement through the link. Instead, when an agent reaches the position of a link, game code needs to react (e.g. through area triggers) and provide means for the agent to move through the link to end up at the links other position (e.g. through teleport or animation). Without that an agent will attempt to move itself along the path of the link. You could end up with an agent walking over a bottomless pit instead of waiting for a moving platform, or walking through a teleporter and proceeding through a wall.

Navigation link script templates

The following script uses the NavigationServer to create a new navigation link.

2D GDScript

```
extends Node2D

var link_rid: RID
var link_start_position: Vector2
var link_end_position: Vector2

func _ready() -> void:
    link_rid = NavigationServer2D.link_create()

    var link_owner_id: int = get_instance_id()
    var link_enter_cost: float = 1.0
    var link_travel_cost: float = 1.0
    var link_navigation_layers: int = 1
    var link_bidirectional: bool = true

    NavigationServer2D.link_set_owner_id(link_rid,
link_owner_id)
    NavigationServer2D.link_set_enter_cost(link_rid,
link_enter_cost)
    NavigationServer2D.link_set_travel_cost(link_rid,
link_travel_cost)
    NavigationServer2D.link_set_navigation_layers(link_rid,
link_navigation_layers)
    NavigationServer2D.link_set_bidirectional(link_rid,
link_bidirectional)

    # Enable the link and set it to the default navigation map.
    NavigationServer2D.link_set_enabled(link_rid, true)
    NavigationServer2D.link_set_map(link_rid,
get_world_2d().get_navigation_map())

    # Move the 2 link positions to their intended global
positions.
    NavigationServer2D.link_set_start_position(link_rid,
link_start_position)
    NavigationServer2D.link_set_end_position(link_rid,
link_end_position)
```

3D GDScript

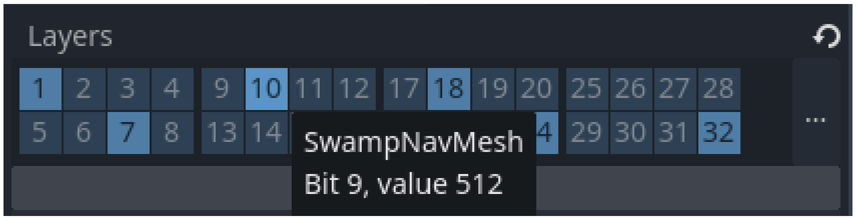
```
extends Node3D

var link_rid: RID
var link_start_position: Vector3
var link_end_position: Vector3

func _ready() -> void:
    link_rid = NavigationServer3D.link_create()

    var link_owner_id: int = get_instance_id()
    var link_enter_cost: float = 1.0
    var link_travel_cost: float = 1.0
    var link_navigation_layers: int = 1
    var link_bidirectional: bool = true

    NavigationServer3D.link_set_owner_id(link_rid,
link_owner_id)
    NavigationServer3D.link_set_enter_cost(link_rid,
link_enter_cost)
    NavigationServer3D.link_set_travel_cost(link_rid,
link_travel_cost)
    NavigationServer3D.link_set_navigation_layers(link_rid,
link_navigation_layers)
    NavigationServer3D.link_set_bidirectional(link_rid,
link_bidirectional)

    # Enable the link and set it to the default navigation map.
    NavigationServer3D.link_set_enabled(link_rid, true)
    NavigationServer3D.link_set_map(link_rid,
get_world_3d().get_navigation_map())

    # Move the 2 link positions to their intended global
positions.
    NavigationServer3D.link_set_start_position(link_rid,
link_start_position)
```

```
    NavigationServer3D.link_set_end_position(link_rid,
link_end_position)
```

Using NavigationLayers

NavigationLayers are an optional feature to further control which navigation meshes are considered in a path query. They work similar to how physics layers control collision between collision objects or how visual layers control what is rendered to the Viewport.

NavigationLayers can be named in the **ProjectSettings** the same as physics layers or visual layers.



If a region has not a single compatible navigation layer with the `navigation_layers` parameter of a path query this regions navigation mesh will be skipped in pathfinding. See [Using NavigationPaths]() for more information on querying the NavigationServer for paths.

NavigationLayers are a single `int` value that is used as a **bitmask**. Many navigation related nodes have `set_navigation_layer_value()` and `get_navigation_layer_value()` functions to set and get a layer number directly without the need for more complex bitwise operations.

In scripts the following helper functions can be used to work with the `navigation_layers` bitmask.

2D GDScript

```
func change_layers():
    var region: NavigationRegion2D =
get_node("NavigationRegion2D")
```

```
    # enables 4-th layer for this region
    region.navigation_layers =
enable_bitmask_inx(region.navigation_layers, 4)
    # disables 1-rst layer for this region
    region.navigation_layers =
disable_bitmask_inx(region.navigation_layers, 1)

    var agent: NavigationAgent2D =
get_node("NavigationAgent2D")
    # make future path queries of this agent ignore regions
with 4-th layer
    agent.navigation_layers =
disable_bitmask_inx(agent.navigation_layers, 4)

    var path_query_navigation_layers: int = 0
    path_query_navigation_layers =
enable_bitmask_inx(path_query_navigation_layers, 2)
    # get a path that only considers 2-nd layer regions
    var path: PoolVector2Array =
NavigationServer2D.map_get_path(
        map,
        start_position,
        target_position,
        true,
        path_query_navigation_layers
        )

static func is_bitmask_inx_enabled(_bitmask: int, _index: int)
-> bool:
    return _bitmask & (1 << _index) != 0

static func enable_bitmask_inx(_bitmask: int, _index: int) ->
int:
    return _bitmask | (1 << _index)

static func disable_bitmask_inx(_bitmask: int, _index: int) ->
int:
    return _bitmask & ~(1 << _index)
```

3D GDScript

```
func change_layers():
    var region: NavigationRegion3D =
get_node("NavigationRegion3D")
    # enables 4-th layer for this region
    region.navigation_layers =
enable_bitmask_inx(region.navigation_layers, 4)
    # disables 1-rst layer for this region
    region.navigation_layers =
disable_bitmask_inx(region.navigation_layers, 1)

    var agent: NavigationAgent3D =
get_node("NavigationAgent3D")
    # make future path queries of this agent ignore regions
with 4-th layer
    agent.navigation_layers =
disable_bitmask_inx(agent.navigation_layers, 4)

    var path_query_navigation_layers: int = 0
    path_query_navigation_layers =
enable_bitmask_inx(path_query_navigation_layers, 2)
    # get a path that only considers 2-nd layer regions
    var path: PoolVector3Array =
NavigationServer3D.map_get_path(
        map,
        start_position,
        target_position,
        true,
        path_query_navigation_layers
        )

static func is_bitmask_inx_enabled(_bitmask: int, _index: int)
-> bool:
    return _bitmask & (1 << _index) != 0

static func enable_bitmask_inx(_bitmask: int, _index: int) ->
int:
    return _bitmask | (1 << _index)

static func disable_bitmask_inx(_bitmask: int, _index: int) ->
int:
    return _bitmask & ~(1 << _index)
```

Changing navigation layers for path queries is a performance friendly alternative to enabling / disabling entire navigation regions. Compared to region changes a navigation path query with different navigation layers does not trigger large scale updates on the NavigationServer.

Changing the navigation layers of NavigationAgent nodes will have an immediate effect on the next path query. Changing the navigation layers of regions will have an effect after the next NavigationServer sync.

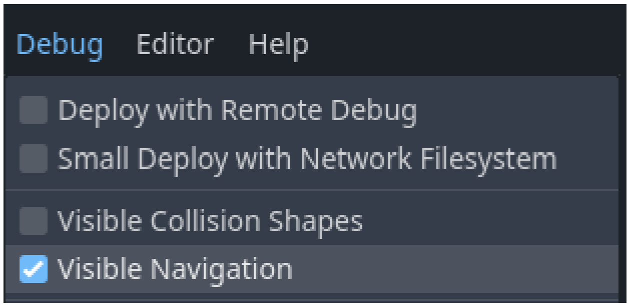
Navigation debug tools

Note

The debug tools, properties and functions are only available in Godot debug builds. Do not use any of them in code that will be part of a release build.

Enabling navigation debug

The navigation debug visualizations are enabled by default inside the editor. To visualize navigation meshes and connections at runtime too, enable the option **Visible Navigation** in the editor **Debug** menu.


Debug
Editor
Help
☐ Deploy with Remote Debug
☐ Small Deploy with Network Filesystem
☐ Visible Collision Shapes
☑ Visible Navigation

In Godot debug builds the navigation debug can also be toggled through the NavigationServer singletons from scripts.

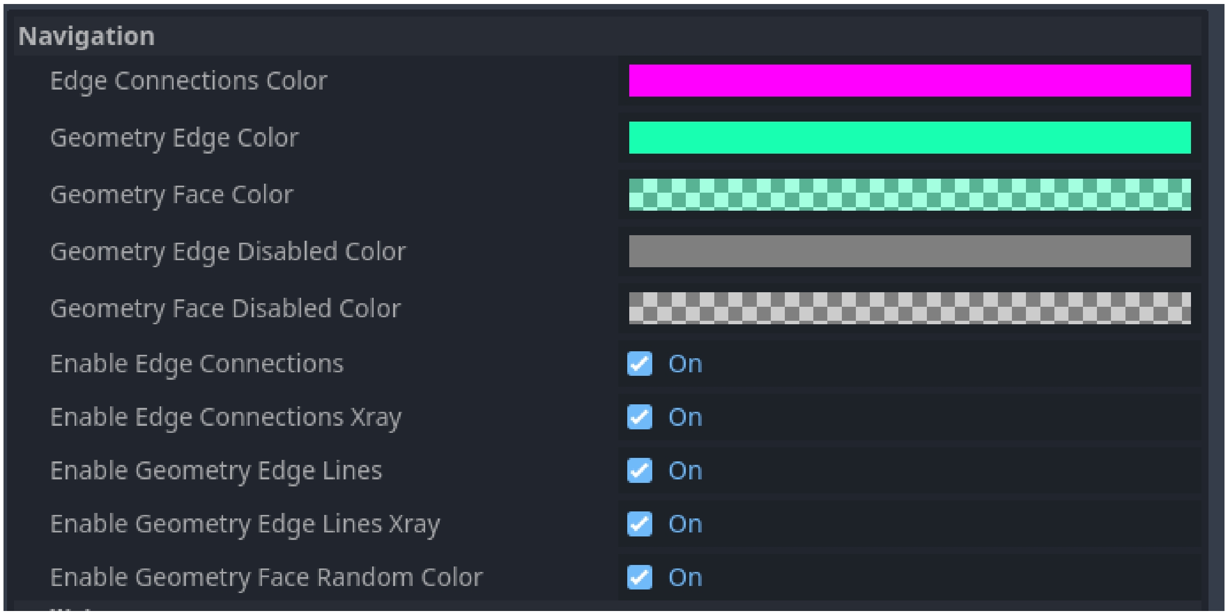
GDScript

```
NavigationServer2D.set_debug_enabled(false)
NavigationServer3D.set_debug_enabled(true)
```

Debug visualizations are currently based on Nodes in the SceneTree. If the NavigationServer2D or NavigationServer3D APIs are used exclusively then changes will not be reflected by the debug navigation tools.

Navigation debug settings

The appearance of navigation debug can be changed in the ProjectSettings under `debug/shapes/navigation`. Certain debug features can also be enabled or disabled at will but may require a scene restart to take effect.



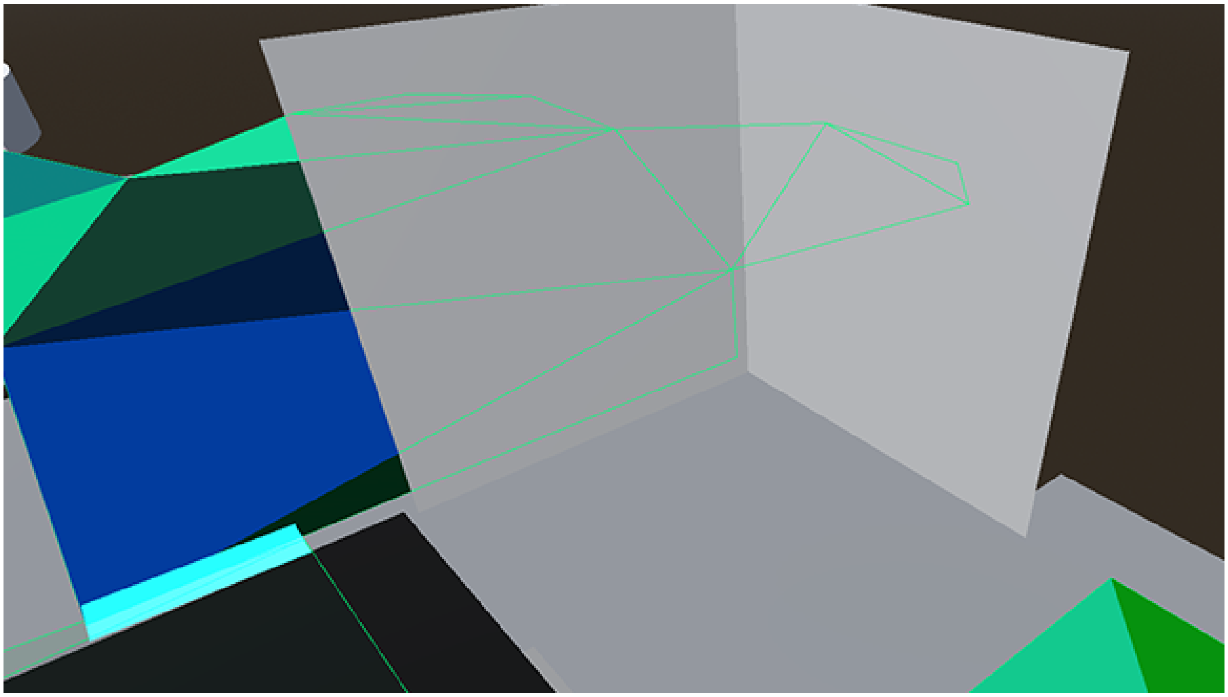
Navigation

Edge Connections Color
Geometry Edge Color
Geometry Face Color
Geometry Edge Disabled Color
Geometry Face Disabled Color
Enable Edge Connections ☑ On
Enable Edge Connections Xray ☑ On
Enable Geometry Edge Lines ☑ On
Enable Geometry Edge Lines Xray ☑ On
Enable Geometry Face Random Color ☑ On

Debug navigation mesh polygons

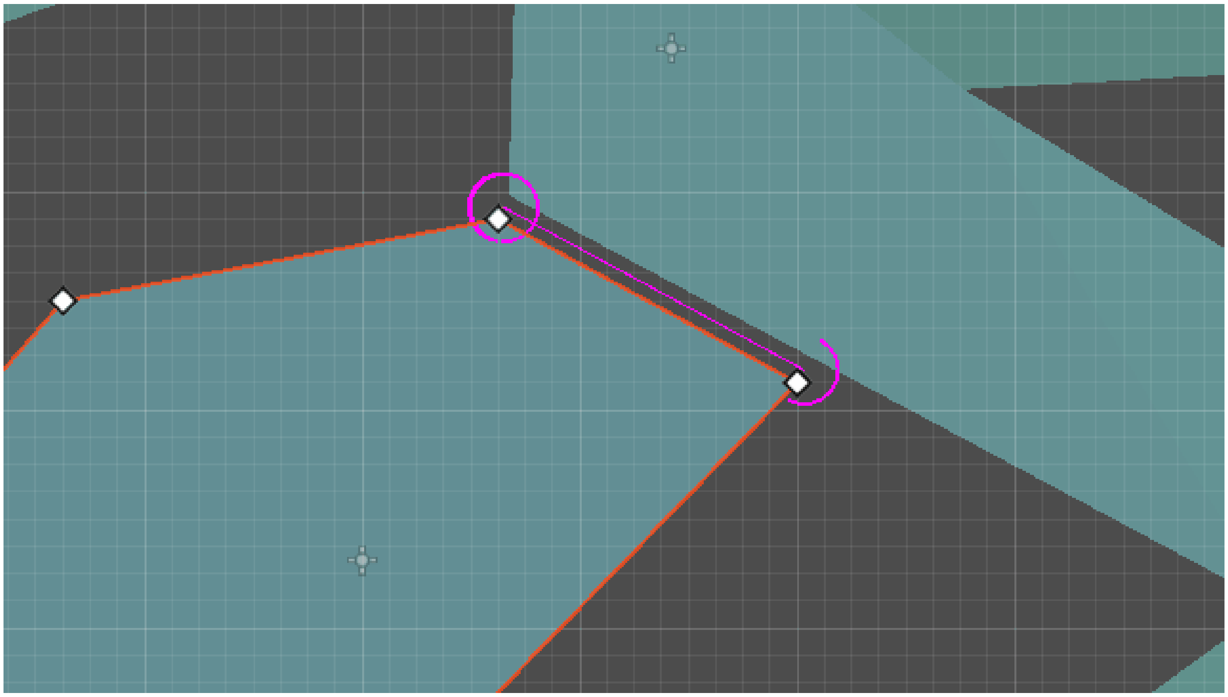
If `enable_edge_lines` is enabled, the edges of navigation mesh polygons will be highlighted. If `enable_edge_lines_xray` is also enabled, the edges of navigation meshes will be visible through geometry.

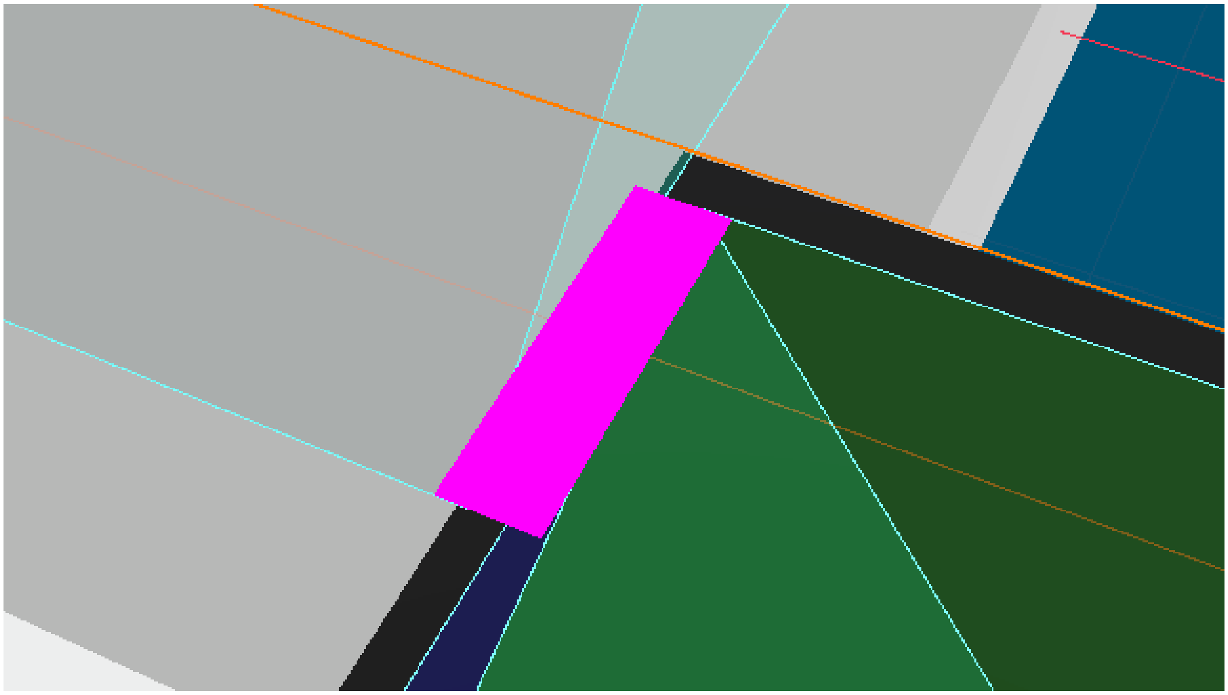
If `enable_geometry_face_random_color` is enabled, the color of each navigation mesh face will be mixed with a random color that is itself mixed with the color specified in `geometry_face_color`.



Debug edge connections

When two navigation meshes are connected within `edge_connection_margin` distance, the connection is overlaid. The color of the overlay is controlled by `edge_connection_color`. The connections can be made visible through geometry with `enable_edge_connections_xray`.



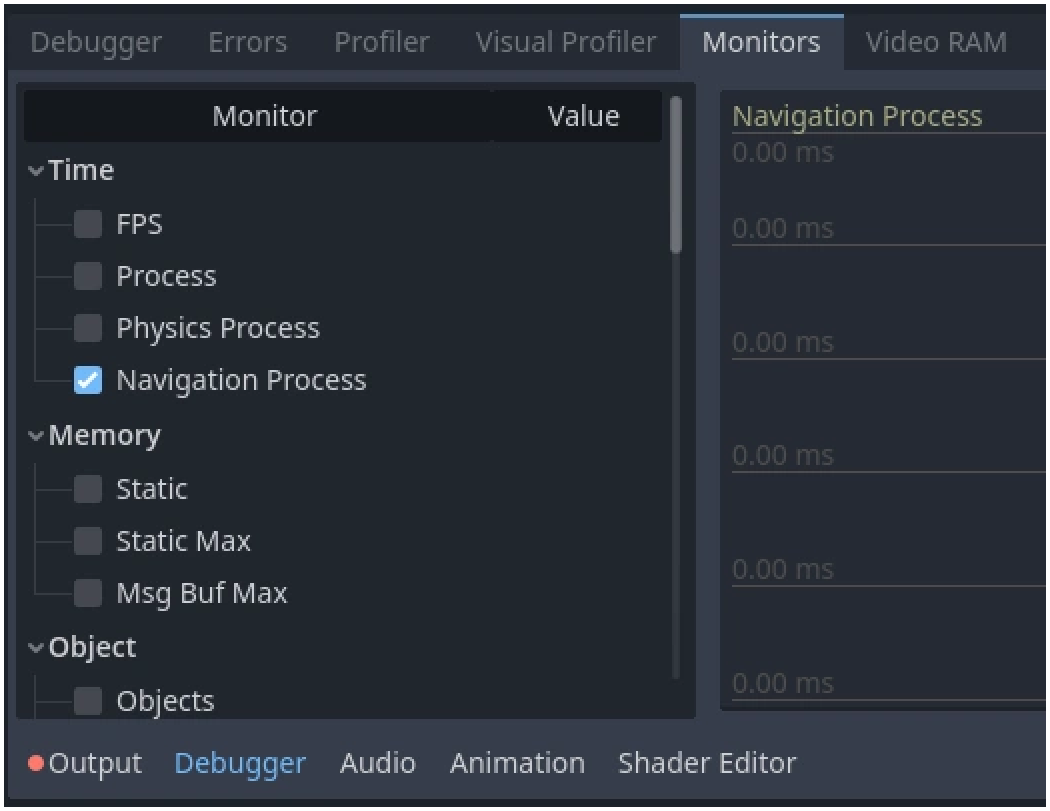


Note

Edge connections are only visible when the NavigationServer is active.

Debug performance

To measure NavigationServer performance a dedicated monitor exists that can be found within the Editor Debugger under *Debugger->Monitors->Navigation Process*.


Debugger
Errors
Profiler
Visual Profiler
Monitors
Video RAM
Monitor
Value
Time
☐ FPS
☐ Process
☐ Physics Process
☑ Navigation Process
Memory
☐ Static
☐ Static Max
☐ Msg Buf Max
Object
☐ Objects
Navigation Process
0.00 ms
0.00 ms
0.00 ms
0.00 ms
0.00 ms
0.00 ms
Output
Debugger
Audio
Animation
Shader Editor

Navigation Process shows how long the NavigationServer spends updating its internals this update frame in milliseconds. Navigation Process works similar to Process for visual frame rendering and Physics Process for collision and fixed updates.

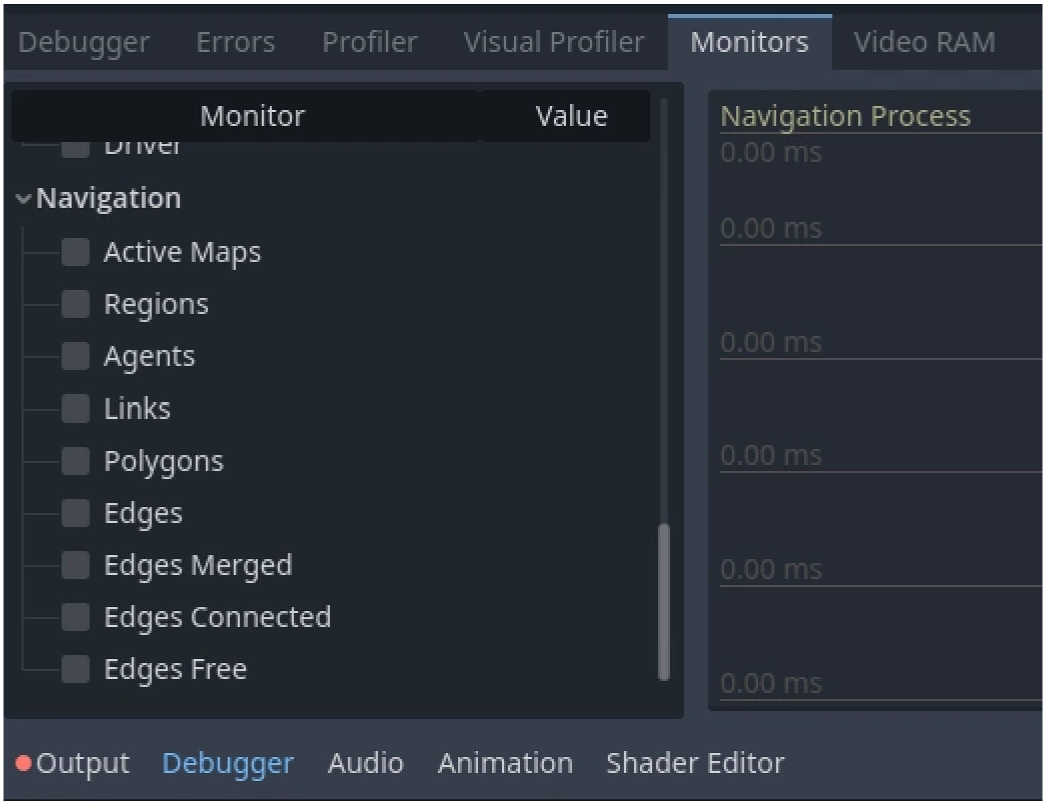
Navigation Process accounts for all updates to **navigation maps**, **navigation regions** and **navigation agents** as well as all the **avoidance calculations** for the update frame.

Note

Navigation Process does NOT include pathfinding performance cause pathfinding operates on the navigation map data independently from the server process update.

Navigation Process should be in general kept as low and as stable as possible for runtime performance to avoid frame rate issues. Note that since the NavigationServer process update happens in the middle of the physics update an increase in Navigation Process will automatically increase Physics Process by the same amount.

Navigation also provides more detailed statistics about the current navigation related objects and navigation map composition on the NavigationServer.


Debugger
Errors
Profiler
Visual Profiler
Monitors
Video RAM
Monitor
Value
Driver
Navigation
Active Maps
Regions
Agents
Links
Polygons
Edges
Edges Merged
Edges Connected
Edges Free
Navigation Process
0.00 ms
0.00 ms
0.00 ms
0.00 ms
0.00 ms
0.00 ms
Output
Debugger
Audio
Animation
Shader Editor

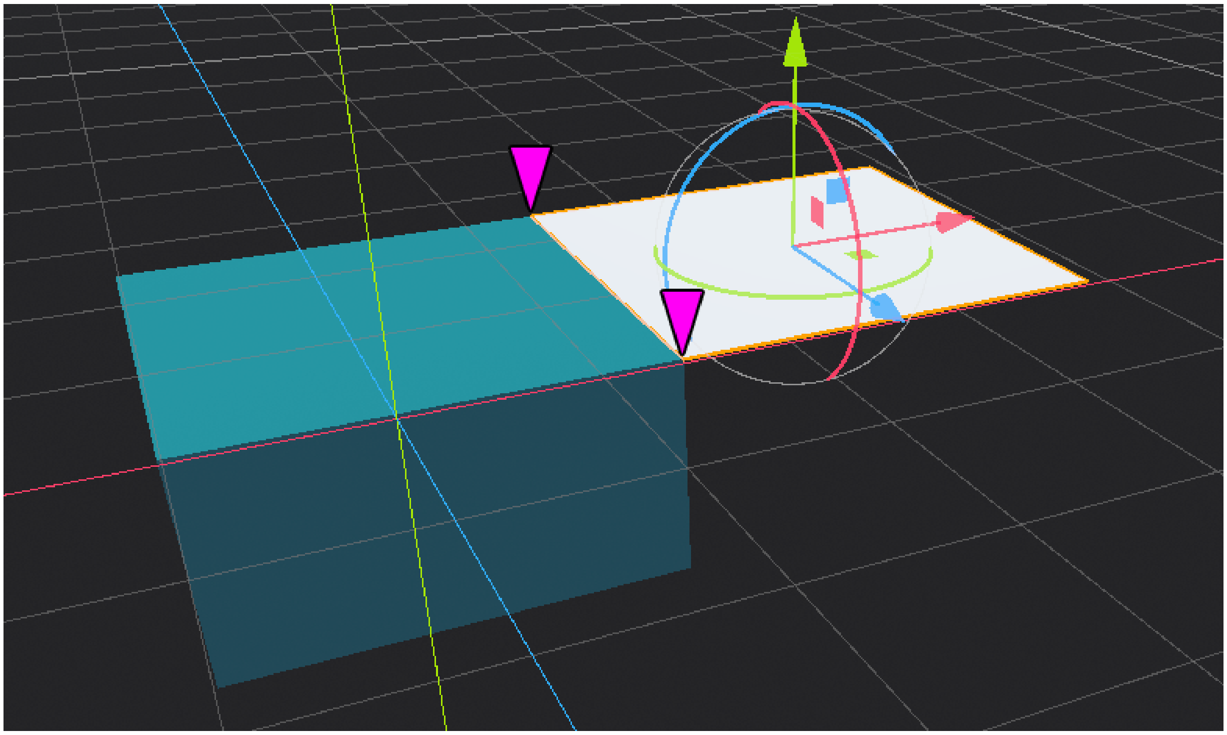
Navigation statistics shown here can not be judged as good or bad for performance as it depends entirely on the project what can be considered as reasonable or horribly excessive.

Navigation statistics help with identifying performance bottlenecks that are less obvious because the source might not always have a visible representation. E.g. pathfinding performance issues created by overly detailed navigation meshes with thousand of edges / polygons or problems caused by procedural navigation gone wrong.

Connecting navigation meshes

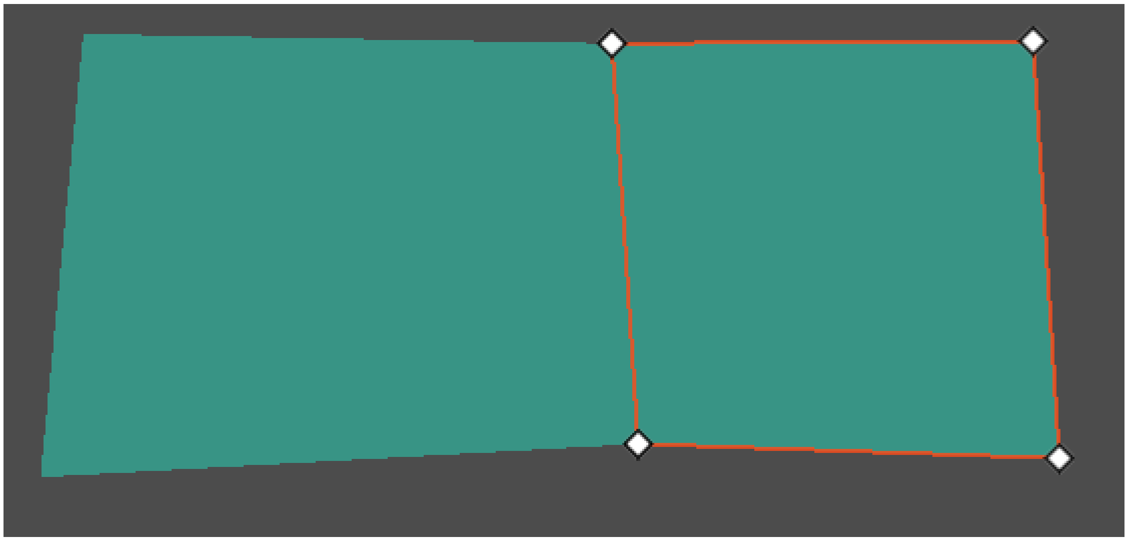
Different NavigationMeshes are automatically merged by the NavigationServer when at least two vertex positions of one edge exactly overlap.

To connect over arbitrary distances see Using NavigationLinks.



The same is true for multiple NavigationPolygon resources. As long as their outline points overlap exactly the NavigationServer will merge them. NavigationPolygon outlines must be from different NavigationPolygon resources to connect.

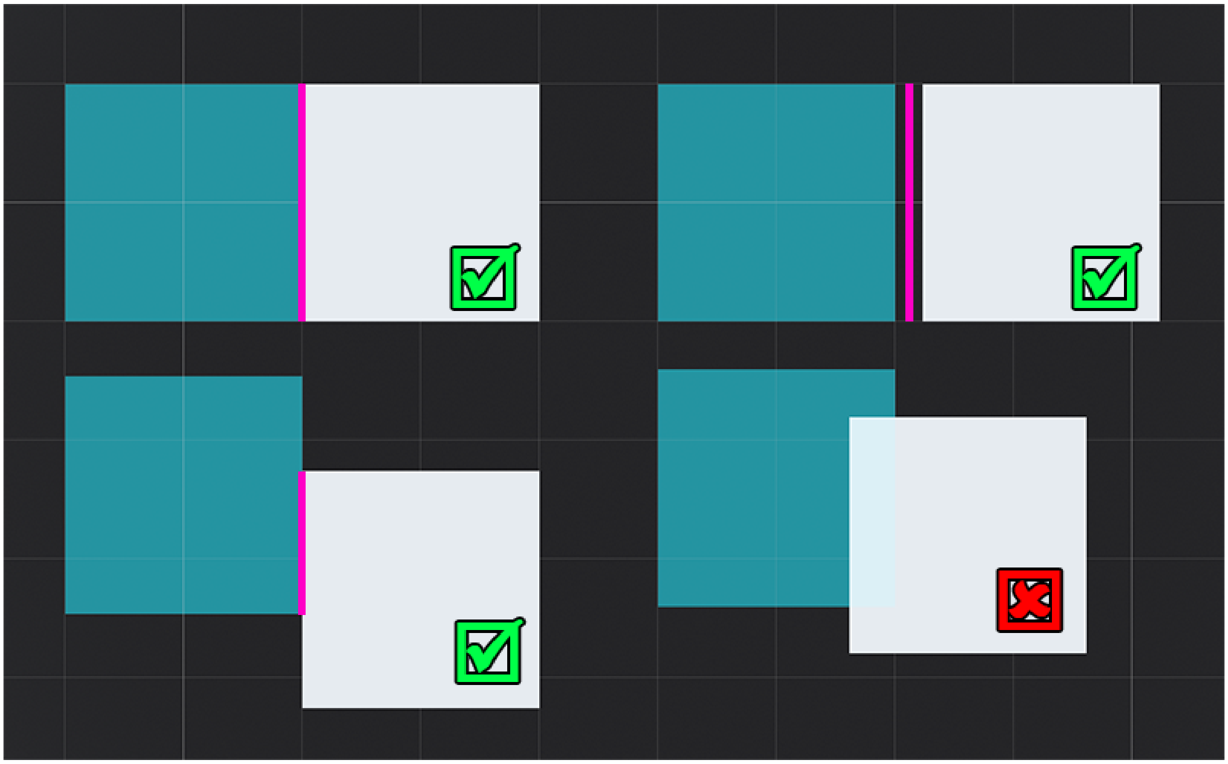
Overlapping or intersecting outlines on the same NavigationPolygon will fail the navigation mesh creation. Overlapping or intersecting outlines from different NavigationPolygons will often fail to create the navigation region edge connections on the NavigationServer and should be avoided.



Warning

Exactly means exactly for the vertex position merge. Small float errors that happen quite regularly with imported meshes will prevent a successful vertex merge.

Alternatively navigation meshes are not merged but still considered as **connected** by the NavigationServer when their edges are nearly parallel and within distance to each other. The connection distance is defined by the `edge_connection_margin` for each navigation map. In many cases navigation mesh edges cannot properly connect when they partly overlap. Better avoid any navigation mesh overlap at all time for a consistent merge behavior.



If navigation debug is enabled and the NavigationServer active the established navigation mesh connections will be visualized. See [Navigation debug tools](#) for more info about navigation debug options.

The default 2D `edge_connection_margin` can be changed in the ProjectSettings under `navigation/2d/default_edge_connection_margin`.

The default 3D `edge_connection_margin` can be changed in the ProjectSettings under `navigation/3d/default_edge_connection_margin`.

The edge connection margin value of any navigation map can also be changed at runtime with the NavigationServer API.

2D GDScript

```
extends Node2D
```

```
func _ready() -> void:
    # 2D margins are designed to work with 2D "pixel" values.
    var default_map_rid: RID =
get_world_2d().get_navigation_map()

NavigationServer2D.map_set_edge_connection_margin(default_map_r
id, 50.0)
```

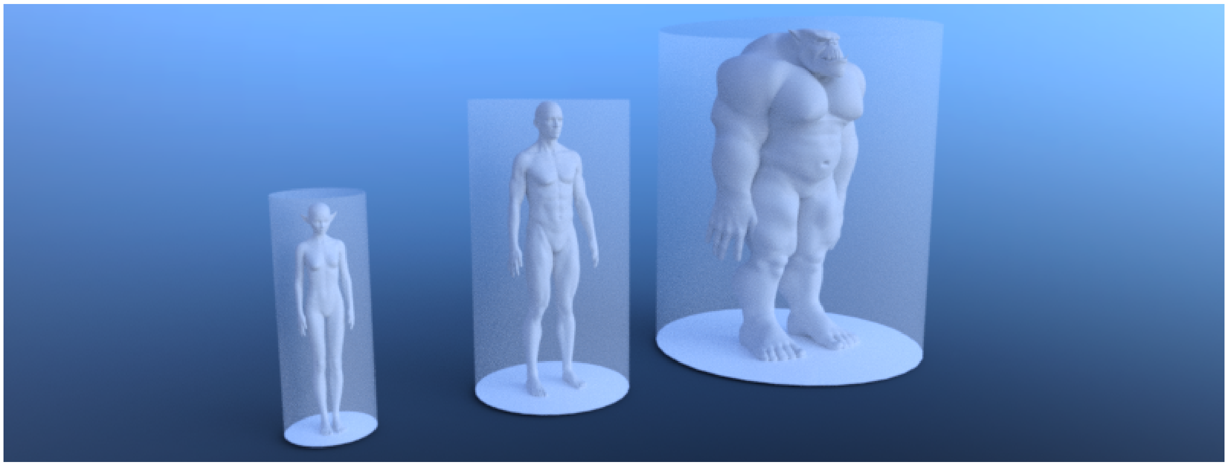
3D GDScript

```
extends Node3D

func _ready() -> void:
    # 3D margins are designed to work with 3D world unit
values.
    var default_map_rid: RID =
get_world_3d().get_navigation_map()

NavigationServer3D.map_set_edge_connection_margin(default_map_r
id, 0.5)
```

Note

Changing the edge connection margin will trigger a full update of all navigation mesh connections on the NavigationServer.

Support different actor types



To support different actor types due to e.g. their sizes each type requires its own navigation map and navigation mesh baked with an appropriated agent radius and height. The same approach can be used to distinguish between e.g. landwalking, swimming or flying agents.

Note

Agents are exclusively defined by a radius and height value for baking navigation meshes, pathfinding and avoidance. More complex shapes are not supported.

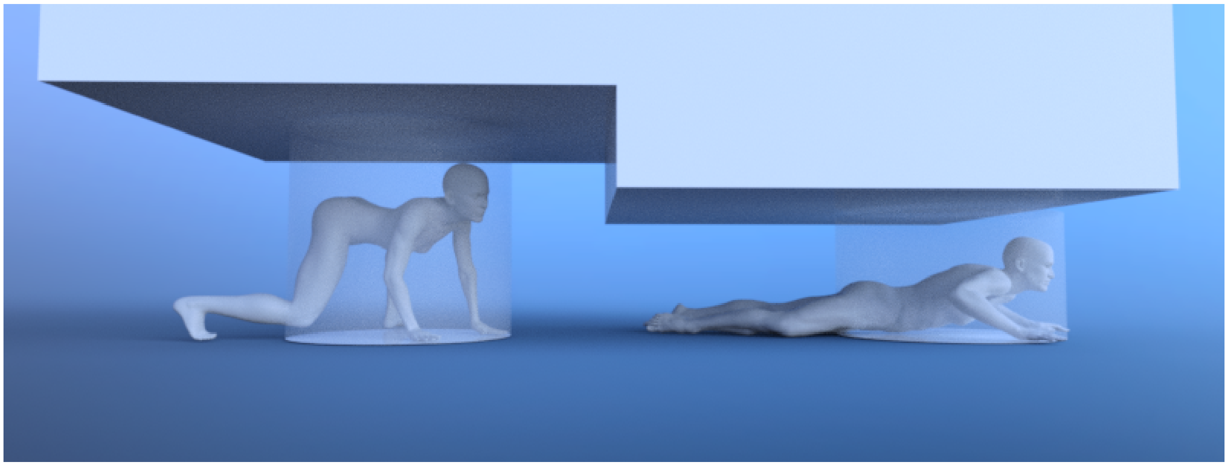
GDScript

```
# Create a navigation mesh resource for each actor size.
var navigation_mesh_standard_size: NavigationMesh =
NavigationMesh.new()
var navigation_mesh_small_size: NavigationMesh =
NavigationMesh.new()
var navigation_mesh_huge_size: NavigationMesh =
NavigationMesh.new()
```

```
# Set appropriated agent parameters.
navigation_mesh_standard_size.agent_radius = 0.5
navigation_mesh_standard_size.agent_height = 1.8
navigation_mesh_small_size.agent_radius = 0.25
navigation_mesh_small_size.agent_height = 0.7
navigation_mesh_huge_size.agent_radius = 1.5
navigation_mesh_huge_size.agent_height = 2.5

# Get the root node to parse geometry for the baking.
var root_node: Node3D =
get_node("NavigationMeshBakingRootNode")

# Create the source geometry resource that will hold the parsed
geometry data.
var source_geometry_data: NavigationMeshSourceGeometryData3D =
NavigationMeshSourceGeometryData3D.new()

# Parse the source geometry from the scene tree on the main
thread.
# The navigation mesh is only required for the parse settings
so any of the three will do.
NavigationServer3D.parse_source_geometry_data(navigation_mesh_s
tandard_size, source_geometry_data, root_node)

# Bake the navigation geometry for each agent size from the
same source geometry.
# If required for performance this baking step could also be
done on background threads.
NavigationServer3D.bake_from_source_geometry_data(navigation_me
sh_standard_size, source_geometry_data)
NavigationServer3D.bake_from_source_geometry_data(navigation_me
sh_small_size, source_geometry_data)
NavigationServer3D.bake_from_source_geometry_data(navigation_me
sh_huge_size, source_geometry_data)

# Create different navigation maps on the NavigationServer.
var navigation_map_standard: RID =
NavigationServer3D.map_create()
var navigation_map_small: RID = NavigationServer3D.map_create()
var navigation_map_huge: RID = NavigationServer3D.map_create()

# Set the new navigation maps as active.
```

```
NavigationServer3D.map_set_active(navigation_map_standard,
true)
NavigationServer3D.map_set_active(navigation_map_small, true)
NavigationServer3D.map_set_active(navigation_map_huge, true)

# Create a region for each map.
var navigation_region_standard: RID =
NavigationServer3D.region_create()
var navigation_region_small: RID =
NavigationServer3D.region_create()
var navigation_region_huge: RID =
NavigationServer3D.region_create()

# Add the regions to the maps.
NavigationServer3D.region_set_map(navigation_region_standard,
navigation_map_standard)
NavigationServer3D.region_set_map(navigation_region_small,
navigation_map_small)
NavigationServer3D.region_set_map(navigation_region_huge,
navigation_map_huge)

# Set navigation mesh for each region.
NavigationServer3D.region_set_navigation_mesh(navigation_region
_standard, navigation_mesh_standard_size)
NavigationServer3D.region_set_navigation_mesh(navigation_region
_small, navigation_mesh_small_size)
NavigationServer3D.region_set_navigation_mesh(navigation_region
_huge, navigation_mesh_huge_size)

# Create start and end position for the navigation path query.
var start_pos: Vector3 = Vector3(0.0, 0.0, 0.0)
var end_pos: Vector3 = Vector3(2.0, 0.0, 0.0)
var use_corridorfunnel: bool = true

# Query paths for each agent size.
var path_standard_agent =
NavigationServer3D.map_get_path(navigation_map_standard,
start_pos, end_pos, use_corridorfunnel)
var path_small_agent =
NavigationServer3D.map_get_path(navigation_map_small,
start_pos, end_pos, use_corridorfunnel)
var path_huge_agent =
NavigationServer3D.map_get_path(navigation_map_huge, start_pos,
end_pos, use_corridorfunnel)
```

Support different actor locomotion



To support different actor locomotion like crouching and crawling, a similar map setup as supporting Support different actor types is required.

Bake different navigation meshes with an appropriate height for crouched or crawling actors so they can find paths through those narrow sections in your game world.

When an actor changes locomotion state, e.g. stands up, starts crouching or crawling, query the appropriate map for a path.

If the avoidance behavior should also change with the locomotion e.g. only avoid while standing or only avoid other agents in the same locomotion state, switch the actor's avoidance agent to another avoidance map with each locomotion change.

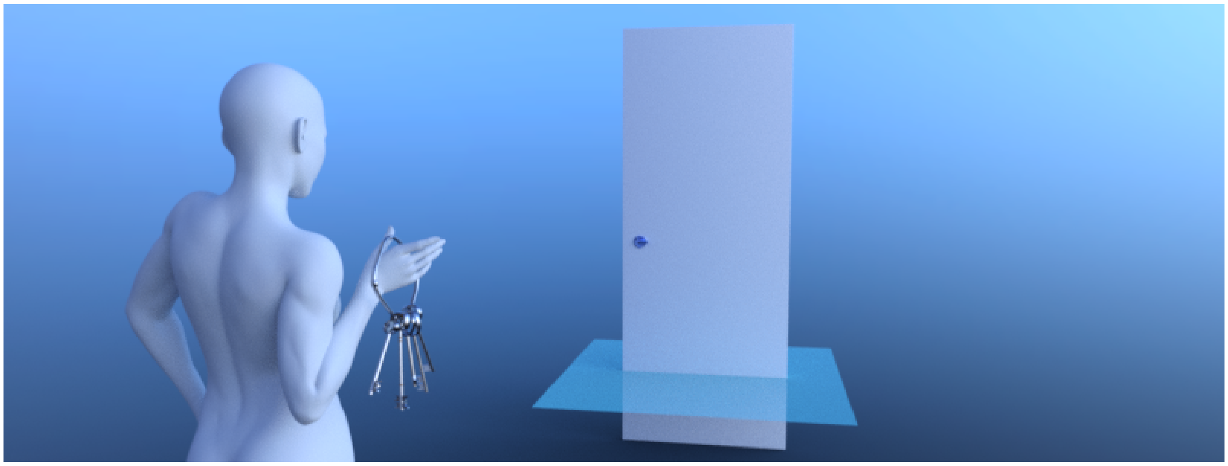
GDScript

```
func update_path():
    if actor_standing:
        path =
NavigationServer3D.map_get_path(standing_navigation_map_rid,
start_position, target_position, true)
    elif actor_crouching:
        path =
NavigationServer3D.map_get_path(crouched_navigation_map_rid,
start_position, target_position, true)
    elif actor_crawling:
        path =
NavigationServer3D.map_get_path(crawling_navigation_map_rid,
start_position, target_position, true)

func change_agent_avoidance_state():

    if actor_standing:
        NavigationServer3D.agent_set_map(avoidance_agent_rid,
standing_navigation_map_rid)
    elif actor_crouching:
        NavigationServer3D.agent_set_map(avoidance_agent_rid,
crouched_navigation_map_rid)
    elif actor_crawling:
        NavigationServer3D.agent_set_map(avoidance_agent_rid,
crawling_navigation_map_rid)
```

Note

While a path query can be execute immediately for multiple maps, the avoidance agent map switch will only take effect after the next server synchronization.

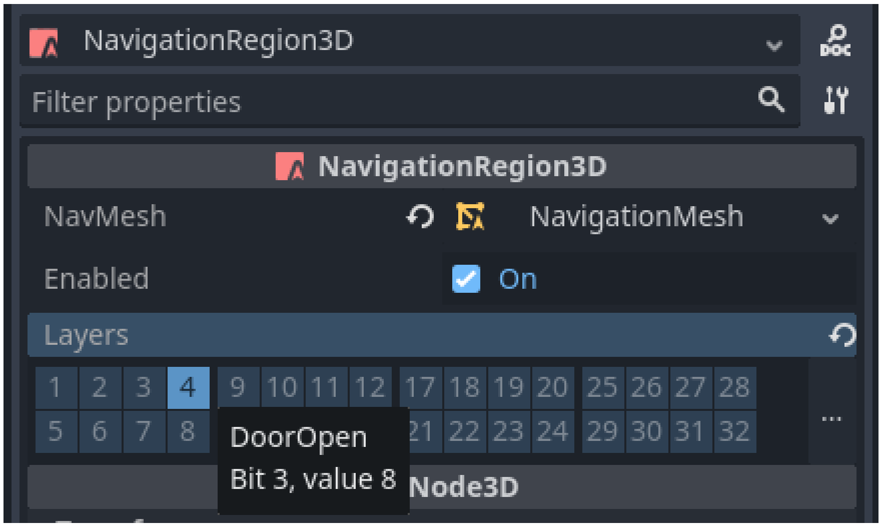
Support different actor area access



A typical example for different area access in gameplay are doors that connect rooms with different navigation meshes and are not accessible by all actors all the time.

Add a NavigationRegion at the door position. Add an appropriate navigation mesh the size of the door that can connect with the surrounding navigation meshes. In order to control access, enable / disable navigation layer bits so path queries that use the same navigation layer bits can find a path through the "door" navigation mesh.

The bitmask can act as a set of door keys or abilities and only actors with at least one matching and enabled bit layer in their pathfinding query will find a path through this region. See Using NavigationLayers for more information on how to work with navigation layers and the bitmask.



NavigationRegion3D
Filter properties
NavigationRegion3D
Layers
1 2 3 4 9 10 11 12 17 18 19 20 25 26 27 28
5 6 7 8 21 22 23 24 29 30 31 32
...
Node3D

NavMesh NavigationMesh

Enabled ☑ On

DoorOpen
Bit 3, value 8

The entire "door" region can also be enabled / disable if required but if disabled will block access for all path queries.

Prefer working with navigation layers in path queries whenever possible as enabling or disabling navigation layers on a region triggers a costly recalculation of the navigation map connections.

Warning

Changing navigation layers will only affect new path queries but not automatically update existing paths.

Optimizing Navigation Performance

Common Navigation related performance problems can be categorized into the following topics:

- Performance problems with parsing scene tree nodes for navigation mesh baking.
- Performance problems with baking the actual navigation mesh.
- Performance problems with NavigationAgent path queries.
- Performance problems with the actual path search.
- Performance problems with synchronizing the navigation map.

In the following sections information can be found on how to identify and fix or at least mitigate their impact on framerates.

Performance problems with parsing scene tree nodes

Tip

Prefer using simple shapes with as few edges as possible e.g. nothing rounded like a circle, sphere or torus.

Prefer using physics collision shapes over complex visual meshes as source geometry as meshes need to be copied from the GPU and are commonly much more detailed than necessary.

In general avoid using very complex geometry as source geometry for baking navigation meshes. E.g. never use a very detailed visual mesh, as parsing its shape to data arrays and voxelizing it for the navigation mesh baking will take a long time for no real quality gain on the final navigation mesh. Instead, use a very simplified level of detail version of a shape. Even better, use very primitive shapes like boxes and rectangles that only roughly cover the same geometry but still yield a baked result good enough for pathfinding.

Prefer using simple physics collision shapes over visual meshes, as the source geometry for baking navigation meshes. Physics shapes are by default very limited and optimized shapes that are easy and quick to parse. A visual mesh on the other hand can range from simple to complex. On top, to gain access to visual mesh data the parser needs to request the mesh data arrays from the RenderingServer as visual mesh data is stored directly on the GPU and is not cached on the CPU. This requires locking the RenderingServer thread and can severely impact framerate at runtime while the rendering runs multi-threaded. If the rendering runs single-threaded, the framerate impact might be even worse and the mesh parsing might freeze the entire game for a few seconds on complex meshes.

Performance problems with navigation mesh baking

Tip

At runtime, always prefer to use a background thread for baking navigation meshes.

Increase NavigationMesh `cell_size` and `cell_height` to create less voxels.

Change the `SamplePartitionType` from watershed to monotone or layers to gain baking performance.

Warning

NEVER scale source geometry with nodes to avoid precision errors. Most scale applies only visually and shapes that are very large at their base scale require still a lot of extra processing even while downscaled.

Baking navigation meshes at runtime should always be done in a background thread if possible. Even small sized navigation meshes can take far longer to bake than what is possible to squeeze into a single frame, at least if the framerate should stay at a bearable level.

Complexity of source geometry data parsed from scene tree nodes has big impact on baking performance as everything needs to be mapped to a grid / voxels. For runtime baking performance the NavigationMesh cell size and cell height should be set as high as possible without causing navigation mesh quality problems for a game. If cell size or cell height is set too low the baking is forced to create an excessive amount of voxels to process the source geometry. If the source geometry spans over a very large game world it is even possible that the baking process runs out off memory in the middle and crashes the game. The partition type can also be lowered depending on how complex the games source geometry is to gain some performance. E.g. games with mostly flat surfaces with blocky geometry can get away with the monotone or layers mode that are a lot faster to bake (e.g. because they require no distance field pass).

Never scale source geometry with nodes. Not only can it result in a lot of precision errors with wrongly matched vertices and edges but also some scaling only exists as visuals and not in the actual parsed data. E.g. if a mesh is downscaled visually in the Editor, e.g. the scale set to 0.001 on a MeshInstance, the mesh still requires a gigantic and very complex voxel grid to be processed for the baking.

Performance problems with NavigationAgent path queries

Tip

Avoid unnecessary path resets and queries every frame in NavigationAgent scripts.

Avoid updating all NavigationAgent paths in the same frame.

Logical errors and wasteful operations in the custom NavigationAgent scripts are very common causes of performance issues, e.g. watch out for resetting the path every single frame. By default NavigationAgents are optimized to only query new paths when the target position changes, the navigation map changes or they are forced too far away from the desired path distance.

E.g. when AI should move to the player, the target position should not be set to the player position every single frame as this queries a new path every frame. Instead, the distance from the current target position to the player position should be compared and only when the player has moved too far away a new target position should be set.

Do not check beforehand if a target position is reachable every frame. What looks like an innocent check is the equivalent of an expensive path query behind the scene. If the plan is to request a new path anyway should the position be reachable, a path should be queried directly. By looking at the last position of the returned path and if that position is in a "reachable" distance to the checked position it answers the "is this position reachable?" question. This avoids doing the equivalent of two full path queries every frame for the same NavigationAgent.

Divide the total number of NavigationAgents into update groups or use random timers so that they do not all request new paths in the same frame.

Performance problems with the actual path search

Tip

Optimize overdetailed navigation meshes by reducing the amount of polygons and edges.

The cost of the actual path search correlates directly with the amount of navigation mesh polygons and edges and not the real size of a game world. If a giant game world uses very optimized navigation meshes with only few polygons that cover large areas, performance should be acceptable. If the game world is splintered into very small navigation meshes that each have tiny polygons (like for TileMaps) pathfinding performance will be reduced.

A common problem is a sudden performance drop when a target position is not reachable in a path query. This performance drop is "normal" and the result of a too large, too unoptimized navigation mesh with way to much polygons and edges to search through. In normal path searches where the target position can be reached quickly the pathfinding will do an early exit as soon as the position is reached which can hide this lack of optimization for a while. If the target position can not be reached the pathfinding has to do a far longer search through the available polygons to confirm that the position is absolutely not reachable.

Performance problems with navigation map synchronization

Tip

Merge navigation meshes polygons by vertex instead of by edge connection wherever possible.

When changes are made to e.g. navigation meshes or navigation regions, the NavigationServer needs to synchronize the navigation map. Depending on the complexity of navigation meshes, this can take a significant amount of time which may impact the framerate.

The NavigationServer merges navigation meshes either by vertex or by edge connection. The merge by vertex happens when the two vertex of two different edges land in the same map grid cells. This is a rather quick and low-cost operation. The merge by edge connection happens in a second pass for all still unmerged edges. All the free edges are checked for possible edge connections by both distance and angle which is rather costly.

So apart from the general rule to have as few polygon edges as possible, as many edges as possible should be merged by vertex upfront so only a few edges are left for the more costly edge connection calculation. The debug Navigation PerformanceMonitor can be used to get statistics on how many polygons and edges are available and how many of them are unmerged or not merged by vertex. If the ratio between vertex merged and edge connections is way off (vertex should be significantly higher) the navigation meshes are properly created or placed very inefficient.

Networking

- High-level multiplayer
- Making HTTP requests
- HTTP client class
- TLS/SSL certificates
- WebSocket
- WebRTC

High-level multiplayer

High-level vs low-level API

The following explains the differences of high- and low-level networking in Godot as well as some fundamentals. If you want to jump in head-first and add networking to your first nodes, skip to Initializing the network below. But make sure to read the rest later on!

Godot always supported standard low-level networking via UDP, TCP and some higher-level protocols such as HTTP and SSL. These protocols are flexible and can be used for almost anything. However, using them to synchronize game state manually can be a large amount of work. Sometimes that work can't be avoided or is worth it, for example when working with a custom server implementation on the backend. But in most cases, it's worthwhile to consider Godot's high-level networking API, which sacrifices some of the fine-grained control of low-level networking for greater ease of use.

This is due to the inherent limitations of the low-level protocols:

- TCP ensures packets will always arrive reliably and in order, but latency is generally higher due to error correction. It's also quite a complex protocol because it understands what a "connection" is, and optimizes for goals that often don't suit applications like multiplayer games. Packets are buffered to be sent in larger batches, trading less per-packet overhead for higher latency. This can be useful for things like HTTP, but generally not for games. Some of this can be configured and disabled (e.g. by disabling "Nagle's algorithm" for the TCP connection).
- UDP is a simpler protocol, which only sends packets (and has no concept of a "connection"). No error correction makes it pretty

quick (low latency), but packets may be lost along the way or received in the wrong order. Added to that, the MTU (maximum packet size) for UDP is generally low (only a few hundred bytes), so transmitting larger packets means splitting them, reorganizing them and retrying if a part fails.

In general, TCP can be thought of as reliable, ordered, and slow; UDP as unreliable, unordered and fast. Because of the large difference in performance, it often makes sense to re-build the parts of TCP wanted for games (optional reliability and packet order), while avoiding the unwanted parts (congestion/traffic control features, Nagle's algorithm, etc). Due to this, most game engines come with such an implementation, and Godot is no exception.

In summary, you can use the low-level networking API for maximum control and implement everything on top of bare network protocols or use the high-level API based on SceneTree that does most of the heavy lifting behind the scenes in a generally optimized way.

Note

Most of Godot's supported platforms offer all or most of the mentioned high- and low-level networking features. As networking is always largely hardware and operating system dependent, however, some features may change or not be available on some target platforms. Most notably, the HTML5 platform currently offers WebSockets and WebRTC support but lacks some of the higher-level features, as well as raw access to low-level protocols like TCP and UDP.

Note

More about TCP/IP, UDP, and networking: https://gafferongames.com/post/udp_vs_tcp/

Gaffer On Games has a lot of useful articles about networking in Games (here [https://gafferongames.com/categories/game-networking/]), including the comprehensive introduction to networking models in games [https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/].

Warning

Adding networking to your game comes with some responsibility. It can make your application vulnerable if done wrong and may lead to cheats or exploits. It may even allow an attacker to compromise the machines your application runs on and use your servers to send spam, attack others or steal your users' data if they play your game.

This is always the case when networking is involved and has nothing to do with Godot. You can of course experiment, but when you release a networked application, always take care of any possible security concerns.

Mid-level abstraction

Before going into how we would like to synchronize a game across the network, it can be helpful to understand how the base network API for synchronization works.

Godot uses a mid-level object MultiplayerPeer. This object is not meant to be created directly, but is designed so that several C++ implementations can provide it.

This object extends from PacketPeer, so it inherits all the useful methods for serializing, sending and receiving data. On top of that, it adds methods to set a peer, transfer mode, etc. It also includes signals that will let you know when peers connect or disconnect.

This class interface can abstract most types of network layers, topologies and libraries. By default, Godot provides an implementation based on ENet (ENetMultiplayerPeer), one based on WebRTC (WebRTCMultiplayerPeer), and one based on WebSocket (WebSocketPeer), but this could be used to implement mobile APIs (for ad hoc WiFi, Bluetooth) or custom device/console-specific networking APIs.

For most common cases, using this object directly is discouraged, as Godot provides even higher level networking facilities. This object is still made available in case a game has specific needs for a lower-level API.

Hosting considerations

When hosting a server, clients on your LAN can connect using the internal IP address which is usually of the form `192.168.*.*`. This internal IP address is **not** reachable by non-LAN/Internet clients.

On Windows, you can find your internal IP address by opening a command prompt and entering `ipconfig`. On macOS, open a Terminal and enter `ifconfig`. On Linux, open a terminal and enter `ip addr`.

If you're hosting a server on your own machine and want non-LAN clients to connect to it, you'll probably have to *forward* the server port on your router. This is required to make your server reachable from the Internet since most residential connections use a NAT [https://en.wikipedia.org/wiki/Network_address_translation]. Godot's high-level multiplayer API only uses UDP, so you must forward the port in UDP, not just TCP.

After forwarding a UDP port and making sure your server uses that port, you can use this website [https://icanhazip.com/] to find your public IP address. Then give this public IP address to any Internet clients that wish to connect to your server.

Godot's high-level multiplayer API uses a modified version of ENet which allows for full IPv6 support.

Initializing the network

High level networking in Godot is managed by the SceneTree.

Each node has a `multiplayer` property, which is a reference to the `MultiplayerAPI` instance configured for it by the scene tree. Initially, every node is configured with the same default `MultiplayerAPI` object.

It is possible to create a new `MultiplayerAPI` object and assign it to a `NodePath` in the the scene tree, which will override `multiplayer` for the node at that path and all of its descendants. This allows sibling nodes to be configured with different peers, which makes it possible to run a server and a client simultaneously in one instance of Godot.

GDScript

```
# By default, these expressions are interchangeable.
multiplayer # Get the MultiplayerAPI object configured for this node.
get_tree().get_multiplayer() # Get the default MultiplayerAPI object.
```

C#

```
// By default, these expressions are interchangeable.
Multiplayer; // Get the MultiplayerAPI object configured for this node.
```

```
GetTree().GetMultiplayer(); // Get the default MultiplayerAPI
object.
```

To initialize networking, a `MultiplayerPeer` object must be created, initialized as a server or client, and passed to the `MultiplayerAPI`.

GDScript

```
# Create client.
var peer = ENetMultiplayerPeer.new()
peer.create_client(IP_ADDRESS, PORT)
multiplayer.multiplayer_peer = peer

# Create server.
var peer = ENetMultiplayerPeer.new()
peer.create_server(PORT, MAX_CLIENTS)
multiplayer.multiplayer_peer = peer
```

C#

```
// Create client.
var peer = new ENetMultiplayerPeer();
peer.CreateClient(IPAddress, Port);
Multiplayer.MultiplayerPeer = peer;

// Create server.
var peer = new ENetMultiplayerPeer();
peer.CreateServer(Port, MaxClients);
Multiplayer.MultiplayerPeer = peer;
```

To terminate networking:

GDScript

```
multiplayer.multiplayer_peer = null
```

C#

```
Multiplayer.MultiplayerPeer = null;
```

Warning

When exporting to Android, make sure to enable the `INTERNET` permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Managing connections

Every peer is assigned a unique ID. The server's ID is always 1, and clients are assigned a random positive integer.

Responding to connections or disconnections is possible by connecting to `MultiplayerAPI`'s signals:

- `peer_connected(id: int)` This signal is emitted with the newly connected peer's ID on each other peer, and on the new peer multiple times, once with each other peer's ID.
- `peer_disconnected(id: int)` This signal is emitted on every remaining peer when one disconnects.

The rest are only emitted on clients:

- `connected_to_server()`
- `connection_failed()`
- `server_disconnected()`

To get the unique ID of the associated peer:

GDScript

```
multiplayer.get_unique_id()
```

C#

```
Multiplayer.GetUniqueId();
```

To check whether the peer is server or client:

GDScript

```
multiplayer.is_server()
```

C#

```
Multiplayer.IsServer();
```

Remote procedure calls

Remote procedure calls, or RPCs, are functions that can be called on other peers. To create one, use the `@rpc` annotation before a function definition. To call an RPC, use `Callable`'s method `rpc()` to call in every peer, or `rpc_id()` to call in a specific peer.

GDScript

```
func _ready():
    if multiplayer.is_server():
        print_once_per_client.rpc()

@rpc
func print_once_per_client():
    print("I will be printed to the console once per each
connected client.")
```

C#

```
public override void _Ready()
{
    if (Multiplayer.IsServer())
    {
        Rpc(MethodName.PrintOncePerClient);
    }
}
```

```
[Rpc]
private void PrintOncePerClient()
{
    GD.Print("I will be printed to the console once per each
connected client.");
}
```

RPCs will not serialize objects or callables.

For a remote call to be successful, the sending and receiving node need to have the same `NodePath`, which means they must have the same name. When using `add_child()` for nodes which are expected to use RPCs, set the argument `force_readable_name` to `true`.

Warning

If a function is annotated with `@rpc` on the client script (resp. server script), then this function must also be declared on the server script (resp. client script). Both RPCs must have the same signature which is evaluated with a checksum of **all RPCs**. All RPCs in a script are checked at once, and all RPCs must be declared on both the client scripts and the server scripts, **even functions that are currently not in use**.

The signature of the RPC includes the `@rpc()` declaration, the function, return type, **and** the NodePath. If an RPC resides in a script attached to `/root/Main/Node1`, then it must reside in precisely the same path and node on both the client script and the server script. Function arguments are not checked for matching between the server and client code (example: `func sendstuff():` and `func sendstuff(arg1, arg2):` **will pass** signature matching).

If these conditions are not met (if all RPCs do not pass signature matching), the script may print an error or cause unwanted behavior. The error message may be unrelated to the RPC function you are currently building and testing.

See further explanation and troubleshooting on this post [https://github.com/godotengine/godot/issues/57869#issuecomment-1034215138].

The annotation can take a number of arguments, which have default values. `@rpc` is equivalent to:

GDScript

```
@rpc("authority", "call_remote", "unreliable", 0)
```

C#

```
[Rpc(MultiplayerApi.RpcMode.Authority, CallLocal = false,
TransferMode = MultiplayerPeer.TransferModeEnum.Unreliable,
TransferChannel = 0)]
```

The parameters and their functions are as follows:

`mode`:

- `"authority"`: Only the multiplayer authority can call remotely. The authority is the server by default, but can be changed per-node using Node.set_multiplayer_authority.
- `"any_peer"`: Clients are allowed to call remotely. Useful for transferring user input.

`sync`:

- `"call_remote"`: The function will not be called on the local peer.
- `"call_local"`: The function can be called on the local peer. Useful when the server is also a player.

`transfer_mode`:

- `"unreliable"` Packets are not acknowledged, can be lost, and can arrive at any order.
- `"unreliable_ordered"` Packets are received in the order they were sent in. This is achieved by ignoring packets that arrive later if another that was sent after them has already been received. Can cause packet loss if used incorrectly.
- `"reliable"` Resend attempts are sent until packets are acknowledged, and their order is preserved. Has a significant performance penalty.

`transfer_channel` is the channel index.

The first 3 can be passed in any order, but `transfer_channel` must always be last.

The function `multiplayer.get_remote_sender_id()` can be used to get the unique id of an rpc sender, when used within the function called by rpc.

GDScript

```
func _on_some_input(): # Connected to some input.
    transfer_some_input.rpc_id(1) # Send the input only to the server.


# Call local is required if the server is also a player.
@rpc("any_peer", "call_local", "reliable")
func transfer_some_input():
    # The server knows who sent the input.
    var sender_id = multiplayer.get_remote_sender_id()
    # Process the input and affect game logic.
```

C#

```
private void OnSomeInput() // Connected to some input.
{
    RpcId(1, MethodName.TransferSomeInput); // Send the input only to the server.
}

// Call local is required if the server is also a player.
[Rpc(MultiplayerApi.RpcMode.AnyPeer, CallLocal = true,
TransferMode = MultiplayerPeer.TransferModeEnum.Reliable)]
private void TransferSomeInput()
{
    // The server knows who sent the input.
    int senderId = Multiplayer.GetRemoteSenderId();
    // Process the input and affect game logic.
}
```

Channels

Modern networking protocols support channels, which are separate connections within the connection. This allows for multiple streams of packets that do not interfere with each other.

For example, game chat related messages and some of the core gameplay messages should all be sent reliably, but a gameplay message should not wait for a chat message to be acknowledged. This can be achieved by using different channels.

Channels are also useful when used with the unreliable ordered transfer mode. Sending packets of variable size with this transfer mode can cause packet loss, since packets which are slower to arrive are ignored. Separating them into multiple streams of homogeneous packets by using channels allows ordered transfer with little packet loss, and without the latency penalty caused by reliable mode.

The default channel with index 0 is actually three different channels - one for each transfer mode.

Example lobby implementation

This is an example lobby that can handle peers joining and leaving, notify UI scenes through signals, and start the game after all clients have loaded the game scene.

GDScript

```
extends Node

# Autoload named Lobby

# These signals can be connected to by a UI lobby scene or the
game scene.
signal player_connected(peer_id, player_info)
signal player_disconnected(peer_id)
signal server_disconnected

const PORT = 7000
const DEFAULT_SERVER_IP = "127.0.0.1" # IPv4 localhost
const MAX_CONNECTIONS = 20

# This will contain player info for every player,
# with the keys being each player's unique IDs.
var players = {}

# This is the local player info. This should be modified
locally
# before the connection is made. It will be passed to every
other peer.
# For example, the value of "name" can be set to something the
player
# entered in a UI scene.
var player_info = {"name": "Name"}

var players_loaded = 0



func _ready():
    multiplayer.peer_connected.connect(_on_player_connected)

multiplayer.peer_disconnected.connect(_on_player_disconnected)
    multiplayer.connected_to_server.connect(_on_connected_ok)
    multiplayer.connection_failed.connect(_on_connected_fail)

multiplayer.server_disconnected.connect(_on_server_disconnected
)


func join_game(address = ""):
    if address.is_empty():
        address = DEFAULT_SERVER_IP
    var peer = ENetMultiplayerPeer.new()
    var error = peer.create_client(address, PORT)
    if error:
        return error
    multiplayer.multiplayer_peer = peer


func create_game():
    var peer = ENetMultiplayerPeer.new()
    var error = peer.create_server(PORT, MAX_CONNECTIONS)
    if error:
        return error
    multiplayer.multiplayer_peer = peer

    players[1] = player_info
    player_connected.emit(1, player_info)


func remove_multiplayer_peer():
    multiplayer.multiplayer_peer = null


# When the server decides to start the game from a UI scene,
# do Lobby.load_game.rpc(filepath)
@rpc("call_local", "reliable")
func load_game(game_scene_path):
    get_tree().change_scene_to_file(game_scene_path)


# Every peer will call this when they have loaded the game
scene.
@rpc("any_peer", "call_local", "reliable")
func player_loaded():
    if multiplayer.is_server():
        players_loaded += 1
        if players_loaded == players.size():
            $/root/Game.start_game()
            players_loaded = 0


# When a peer connects, send them my player info.
# This allows transfer of all desired data for each player, not only the unique ID.
func _on_player_connected(id):
    _register_player.rpc_id(id, player_info)


@rpc("any_peer", "reliable")
func _register_player(new_player_info):
    var new_player_id = multiplayer.get_remote_sender_id()
    players[new_player_id] = new_player_info
    player_connected.emit(new_player_id, new_player_info)


func _on_player_disconnected(id):
    players.erase(id)
    player_disconnected.emit(id)


func _on_connected_ok():
    var peer_id = multiplayer.get_unique_id()
    players[peer_id] = player_info
    player_connected.emit(peer_id, player_info)


func _on_connected_fail():
    multiplayer.multiplayer_peer = null


func _on_server_disconnected():
    multiplayer.multiplayer_peer = null
    players.clear()
    server_disconnected.emit()
```

C#

```
using Godot;

public partial class Lobby : Node
{
    public static Lobby Instance { get; private set; }

    // These signals can be connected to by a UI lobby scene or
the game scene.
    [Signal]
    public delegate void PlayerConnectedEventHandler(int
peerId, Godot.Collections.Dictionary<string, string>
playerInfo);
    [Signal]
    public delegate void PlayerDisconnectedEventHandler(int
peerId);
    [Signal]
    public delegate void ServerDisconnectedEventHandler();

    private const int Port = 7000;
    private const string DefaultServerIP = "127.0.0.1"; // IPv4
localhost
    private const int MaxConnections = 20;

    // This will contain player info for every player,
    // with the keys being each player's unique IDs.
    private Godot.Collections.Dictionary<long,
Godot.Collections.Dictionary<string, string>> _players = new
Godot.Collections.Dictionary<long,
Godot.Collections.Dictionary<string, string>>();

    // This is the local player info. This should be modified
locally
    // before the connection is made. It will be passed to
every other peer.
    // For example, the value of "name" can be set to something
the player
    // entered in a UI scene.
    private Godot.Collections.Dictionary<string, string>
_playerInfo = new Godot.Collections.Dictionary<string, string>
()
    {
        { "Name", "PlayerName" },
    };
```

```
private int _playersLoaded = 0;

public override void _Ready()
{
    Instance = this;
    Multiplayer.PeerConnected += OnPlayerConnected;
    Multiplayer.PeerDisconnected += OnPlayerDisconnected;
    Multiplayer.ConnectedToServer += OnConnectOk;
    Multiplayer.ConnectionFailed += OnConnectionFail;
    Multiplayer.ServerDisconnected += OnServerDisconnected;
}

private Error JoinGame(string address = "")
{
    if (string.IsNullOrEmpty(address))
    {
        address = DefaultServerIP;
    }

    var peer = new ENetMultiplayerPeer();
    Error error = peer.CreateClient(address, Port);

    if (error != Error.Ok)
    {
        return error;
    }

    Multiplayer.MultiplayerPeer = peer;
    return Error.Ok;
}

private Error CreateGame()
{
    var peer = new ENetMultiplayerPeer();
    Error error = peer.CreateServer(Port, MaxConnections);

    if (error != Error.Ok)
    {
        return error;
    }

    Multiplayer.MultiplayerPeer = peer;
```

```
        _players[1] = _playerInfo;
        EmitSignal(SignalName.PlayerConnected, 1, _playerInfo);
        return Error.Ok;
    }

    private void RemoveMultiplayerPeer()
    {
        Multiplayer.MultiplayerPeer = null;
    }

    // When the server decides to start the game from a UI
scene,
    // do Rpc(Lobby.MethodName.LoadGame, filePath);
    [Rpc(CallLocal = true,TransferMode =
MultiplayerPeer.TransferModeEnum.Reliable)]
    private void LoadGame(string gameScenePath)
    {
        GetTree().ChangeSceneToFile(gameScenePath);
    }

    // Every peer will call this when they have loaded the game
scene.
    [Rpc(MultiplayerApi.RpcMode.AnyPeer,CallLocal =
true,TransferMode = MultiplayerPeer.TransferModeEnum.Reliable)]
    private void PlayerLoaded()
    {
        if (Multiplayer.IsServer())
        {
            _playersLoaded += 1;
            if (_playersLoaded == _players.Count)
            {
                GetNode<Game>("/root/Game").StartGame();
                _playersLoaded = 0;
            }
        }
    }

    // When a peer connects, send them my player info.
    // This allows transfer of all desired data for each
player, not only the unique ID.
    private void OnPlayerConnected(long id)
    {
        RpcId(id, MethodName.RegisterPlayer, _playerInfo);
    }

    [Rpc(MultiplayerApi.RpcMode.AnyPeer,TransferMode =
MultiplayerPeer.TransferModeEnum.Reliable)]
    private void
RegisterPlayer(Godot.Collections.Dictionary<string, string>
newPlayerInfo)
    {
        int newPlayerId = Multiplayer.GetRemoteSenderId();
        _players[newPlayerId] = newPlayerInfo;
        EmitSignal(SignalName.PlayerConnected, newPlayerId,
newPlayerInfo);
    }

    private void OnPlayerDisconnected(long id)
    {
        _players.Remove(id);
        EmitSignal(SignalName.PlayerDisconnected, id);
    }

    private void OnConnectOk()
    {
        int peerId = Multiplayer.GetUniqueId();
        _players[peerId] = _playerInfo;
        EmitSignal(SignalName.PlayerConnected, peerId,
_playerInfo);
    }

    private void OnConnectionFail()
    {
        Multiplayer.MultiplayerPeer = null;
    }

    private void OnServerDisconnected()
    {
        Multiplayer.MultiplayerPeer = null;
        _players.Clear();
        EmitSignal(SignalName.ServerDisconnected);
    }
}
```

The game scene's root node should be named Game. In the script attached to it:

GDScript

```gdscript
extends Node3D # Or Node2D.



func _ready():
    # Preconfigure game.

    Lobby.player_loaded.rpc_id(1) # Tell the server that this peer has loaded.


# Called only on the server.
func start_game():
    # All peers are ready to receive RPCs in this scene.
```

C#

```csharp
using Godot;

public partial class Game : Node3D // Or Node2D.
{
    public override void _Ready()
    {
        // Preconfigure game.

        Lobby.Instance.RpcId(1, Lobby.MethodName.PlayerLoaded); // Tell the server that this peer has loaded.
    }

    // Called only on the server.
    public void StartGame()
    {
        // All peers are ready to receive RPCs in this scene.
    }
}
```

Exporting for dedicated servers

Once you've made a multiplayer game, you may want to export it to run it on a dedicated server with no GPU available. See Exporting for dedicated servers for more information.

Note

The code samples on this page aren't designed to run on a dedicated server. You'll have to modify them so the server isn't considered to be a player. You'll also have to modify the game starting mechanism so that the first player who joins can start the game.

Making HTTP requests

Why use HTTP?

HTTP requests [https://developer.mozilla.org/en-US/docs/Web/HTTP] are useful to communicate with web servers and other non-Godot programs.

Compared to Godot's other networking features (like High-level multiplayer), HTTP requests have more overhead and take more time to get going, so they aren't suited for real-time communication, and aren't great to send lots of small updates as is common for multiplayer gameplay.

HTTP, however, offers interoperability with external web resources and is great at sending and receiving large amounts of data, for example to transfer files like game assets. These assets can then be loaded using runtime file loading and saving.

So HTTP may be useful for your game's login system, lobby browser, to retrieve some information from the web or to download game assets.

This tutorial assumes some familiarity with Godot and the Godot Editor. Refer to the Introduction and the Step by step tutorial, especially its Nodes and Scenes and Creating your first script pages if needed.

HTTP requests in Godot

The HTTPRequest node is the easiest way to make HTTP requests in Godot. It is backed by the more low-level HTTPClient, for which a tutorial is available here.

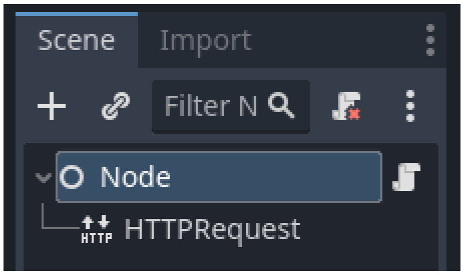
For this example, we will make an HTTP request to GitHub to retrieve the name of the latest Godot release.

Warning

When exporting to **Android**, make sure to enable the **Internet** permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by the Android OS.

Preparing the scene

Create a new empty scene, add a root Node and add a script to it. Then add an HTTPRequest node as a child.



Scripting the request

When the project is started (so in `_ready()`), we're going to send an HTTP request to Github using our HTTPRequest node, and once the request completes, we're going to parse the returned JSON data, look for the `name` field and print that to console.

GDScript

```
extends Node

func _ready():

$HTTPRequest.request_completed.connect(_on_request_completed)

$HTTPRequest.request("https://api.github.com/repos/godotengine/
godot/releases/latest")

func _on_request_completed(result, response_code, headers,
body):
    var json = JSON.parse_string(body.get_string_from_utf8())
    print(json["name"])
```

C#

```
using Godot;
using System.Text;

public partial class MyNode : Node
{
    public override void _Ready()
    {
        HttpRequest httpRequest = GetNode<HttpRequest>
("HTTPRequest");
        httpRequest.RequestCompleted += OnRequestCompleted;

httpRequest.Request("https://api.github.com/repos/godotengine/g
odot/releases/latest");
    }

    private void OnRequestCompleted(long result, long
responseCode, string[] headers, byte[] body)
    {
        Godot.Collections.Dictionary json =
Json.ParseString(Encoding.UTF8.GetString(body)).AsGodotDictiona
ry();
        GD.Print(json["name"]);
    }
}
```

Save the script and the scene, and run the project. The name of the most recent Godot release on Github should be printed to the output log. For more information on parsing JSON, see the class references for JSON.

Note that you may want to check whether the `result` equals `RESULT_SUCCESS` and whether a JSON parsing error occurred, see the JSON class reference and HTTPRequest for more.

You have to wait for a request to finish before sending another one. Making multiple request at once requires you to have one node per request. A common strategy is to create and delete HTTPRequest nodes at runtime as necessary.

Sending data to the server

Until now, we have limited ourselves to requesting data from a server. But what if you need to send data to the server? Here is a common way of doing it:

GDScript

```
var json = JSON.stringify(data_to_send)
var headers = ["Content-Type: application/json"]
$HTTPRequest.request(url, headers, HTTPClient.METHOD_POST,
json)
```

C#

```
string json = Json.Stringify(dataToSend);
string[] headers = new string[] { "Content-Type:
application/json" };
HttpRequest httpRequest = GetNode<HttpRequest>("HTTPRequest");
httpRequest.Request(url, headers, HttpClient.Method.Post,
json);
```

Setting custom HTTP headers

Of course, you can also set custom HTTP headers. These are given as a string array, with each string containing a header in the format `"header: value"`. For example, to set a custom user agent (the HTTP `User-Agent` header) you could use the following:

GDScript

```
$HTTPRequest.request("https://api.github.com/repos/godotengine/
godot/releases/latest", ["User-Agent: YourCustomUserAgent"])
```

C#

```
HttpRequest httpRequest = GetNode<HttpRequest>("HTTPRequest");
httpRequest.Request("https://api.github.com/repos/godotengine/g
odot/releases/latest", new string[] { "User-Agent:
YourCustomUserAgent" });
```

Warning

Be aware that someone might analyse and decompile your released application and thus may gain access to any embedded authorization information like tokens, usernames or passwords. That means it is usually not a good idea to embed things such as database access credentials inside your game. Avoid providing information useful to an attacker whenever possible.

HTTP client class

HTTPClient provides low-level access to HTTP communication. For a higher-level interface, you may want to take a look at HTTPRequest first, which has a tutorial available here.

Warning

When exporting to Android, make sure to enable the `INTERNET` permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Here's an example of using the HTTPClient class. It's just a script, so it can be run by executing:

GDScript

```
c:\godot> godot -s http_test.gd
```

C#

```
c:\godot> godot -s HTTPTest.cs
```

It will connect and fetch a website.

GDScript

```
extends SceneTree

# HTTPClient demo
# This simple class can do HTTP requests; it will not block,
but it needs to be polled.

func _init():
    var err = 0
    var http = HTTPClient.new() # Create the Client.

    err = http.connect_to_host("www.php.net", 80) # Connect to host/port.
    assert(err == OK) # Make sure connection is OK.

    # Wait until resolved and connected.
    while http.get_status() == HTTPClient.STATUS_CONNECTING or http.get_status() == HTTPClient.STATUS_RESOLVING:
        http.poll()
        print("Connecting...")
        await get_tree().process_frame

    assert(http.get_status() == HTTPClient.STATUS_CONNECTED) # Check if the connection was made successfully.

    # Some headers
    var headers = [
        "User-Agent: Pirulo/1.0 (Godot)",
        "Accept: */*"
    ]

    err = http.request(HTTPClient.METHOD_GET, "/ChangeLog-5.php", headers) # Request a page from the site (this one was chunked..)
    assert(err == OK) # Make sure all is OK.

    while http.get_status() == HTTPClient.STATUS_REQUESTING:
        # Keep polling for as long as the request is being processed.
        http.poll()
        print("Requesting...")
        await get_tree().process_frame

    assert(http.get_status() == HTTPClient.STATUS_BODY or http.get_status() == HTTPClient.STATUS_CONNECTED) # Make sure request finished well.

    print("response? ", http.has_response()) # Site might not have a response.

    if http.has_response():
```

C#

```
        # If there is a response...

        headers = http.get_response_headers_as_dictionary() # Get response headers.
        print("code: ", http.get_response_code()) # Show response code.
        print("**headers:\\n", headers) # Show headers.

        # Getting the HTTP Body

        if http.is_response_chunked():
            # Does it use chunks?
            print("Response is Chunked!")
        else:
            # Or just plain Content-Length
            var bl = http.get_response_body_length()
            print("Response Length: ", bl)

        # This method works for both anyway

        var rb = PackedByteArray() # Array that will hold the data.

        while http.get_status() == HTTPClient.STATUS_BODY:
            # While there is body left to be read
            http.poll()
            # Get a chunk.
            var chunk = http.read_response_body_chunk()
            if chunk.size() == 0:
                await get_tree().process_frame
            else:
                rb = rb + chunk # Append to read buffer.
        # Done!

        print("bytes got: ", rb.size())
        var text = rb.get_string_from_ascii()
        print("Text: ", text)

    quit()
```

C#

```
using Godot;

public partial class HTTPTest : SceneTree
{
    // HTTPClient demo.
    // This simple class can make HTTP requests; it will not
block, but it needs to be polled.
    public override async void _Initialize()
    {
        Error err;
        HTTPClient http = new HTTPClient(); // Create the
client.

        err = http.ConnectToHost("www.php.net", 80); // Connect
to host/port.
        Debug.Assert(err == Error.Ok); // Make sure the
connection is OK.

        // Wait until resolved and connected.
        while (http.GetStatus() == HTTPClient.Status.Connecting
|| http.GetStatus() == HTTPClient.Status.Resolving)
        {
            http.Poll();
            GD.Print("Connecting...");
            OS.DelayMsec(500);
        }

        Debug.Assert(http.GetStatus() ==
HTTPClient.Status.Connected); // Check if the connection was
made successfully.

        // Some headers.
        string[] headers = { "User-Agent: Pirulo/1.0 (Godot)",
"Accept: */*" };

        err = http.Request(HTTPClient.Method.Get, "/ChangeLog-
5.php", headers); // Request a page from the site.
        Debug.Assert(err == Error.Ok); // Make sure all is OK.

        // Keep polling for as long as the request is being
processed.
        while (http.GetStatus() ==
HTTPClient.Status.Requesting)
```

```
        {
            http.Poll();
            GD.Print("Requesting...");
            if (OS.HasFeature("web"))
            {
                // Synchronous HTTP requests are not supported
on the web,
                // so wait for the next main loop iteration.
                await ToSignal(Engine.GetMainLoop(),
"idle_frame");
            }
            else
            {
                OS.DelayMsec(500);
            }
        }

        Debug.Assert(http.GetStatus() == HTTPClient.Status.Body
|| http.GetStatus() == HTTPClient.Status.Connected); // Make
sure the request finished well.

        GD.Print("Response? ", http.HasResponse()); // The site
might not have a response.

        // If there is a response...
        if (http.HasResponse())
        {
            headers = http.GetResponseHeaders(); // Get
response headers.
            GD.Print("Code: ", http.GetResponseCode()); // Show
response code.
            GD.Print("Headers:");
            foreach (string header in headers)
            {
                // Show headers.
                GD.Print(header);
            }

            if (http.IsResponseChunked())
            {
                // Does it use chunks?
                GD.Print("Response is Chunked!");
            }
            else
            {
                // Or just Content-Length.
                GD.Print("Response Length: ",
http.GetResponseBodyLength());
            }

            // This method works for both anyways.
            List<byte> rb = new List<byte>(); // List that will
hold the data.

            // While there is data left to be read...
            while (http.GetStatus() == HTTPClient.Status.Body)
            {
                http.Poll();
                byte[] chunk = http.ReadResponseBodyChunk(); //
Read a chunk.
                if (chunk.Length == 0)
                {
                    // If nothing was read, wait for the buffer
to fill.
                    OS.DelayMsec(500);
                }
                else
                {
                    // Append the chunk to the read buffer.
                    rb.AddRange(chunk);
                }
            }

            // Done!
            GD.Print("Bytes Downloaded: ", rb.Count);
            string text =
Encoding.ASCII.GetString(rb.ToArray());
            GD.Print(text);
        }
        Quit();
    }
}
```

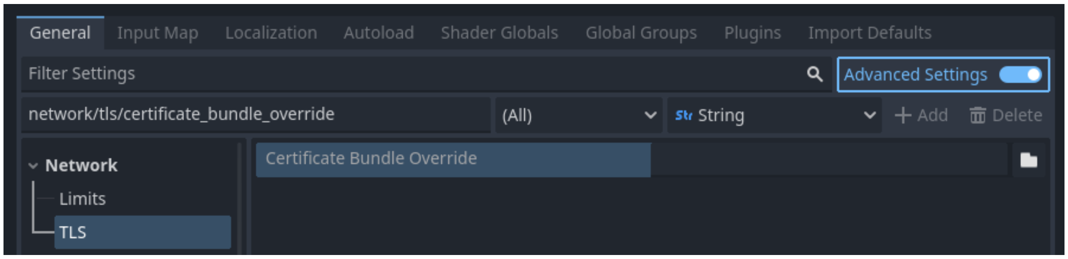
TLS/SSL certificates

Introduction

It is often desired to use TLS connections (also known as SSL connections) for communications to avoid "man in the middle" attacks. Godot has a connection wrapper, StreamPeerTLS, which can take a regular connection and add security around it. The HTTPClient and HTTPRequest classes also support HTTPS using this same wrapper.

Godot will try to use the TLS certificate bundle provided by the operating system, but also includes the TLS certificate bundle from Mozilla [https://github.com/godotengine/godot/blob/master/thirdparty/certs/ca-certificates.crt] as a fallback.

You can alternatively force your own certificate bundle in the Project Settings:


General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Import Defaults
Filter Settings
Advanced Settings
network/tls/certificate_bundle_override
(All)
String
Add
Delete
Network
Limits
TLS
Certificate Bundle Override

Setting the TLS certificate bundle override project setting

When set, this file *overrides* the operating system provided bundle by default. This file should contain any number of public certificates in PEM format [https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail].

There are two ways to obtain certificates:

Obtain a certificate from a certificate authority

The main approach to getting a certificate is to use a certificate authority (CA) such as Let's Encrypt [https://letsencrypt.org/]. This is a more cumbersome process than a self-signed certificate, but it's more "official" and ensures your identity is clearly represented. The resulting certificate is also trusted by applications such as web browsers, unlike a self-signed certificate which requires additional configuration on the client side before it's considered trusted.

These certificates do not require any configuration on the client to work, since Godot already bundles the Mozilla certificate bundle in the editor and exported projects.

Generate a self-signed certificate

For most use cases, it's recommended to go through certificate authority as the process is free with certificate authorities such as Let's Encrypt. However, if using a certificate authority is not an option, then you can generate a self-signed certificate and tell the client to consider your self-signed certificate as trusted.

To create a self-signed certificate, generate a private and public key pair and add the public key (in PEM format) to the CRT file specified in the Project Settings.

Warning

The private key should **only** go to your server. The client must not have access to it: otherwise, the security of the certificate will be compromised.

Warning

When specifying a self-signed certificate as TLS bundle in the project settings, normal domain name validation is enforced via the certificate CN and alternative names. See TLSOptions to customize domain name validation.

For development purposes Godot can generate self-signed certificates via Crypto.generate_self_signed_certificate.

Alternatively, OpenSSL has some documentation about generating keys [https://raw.githubusercontent.com/openssl/openssl/master/doc/HOWTO/keys.txt] and certificates [https://raw.githubusercontent.com/openssl/openssl/master/doc/HOWTO/certificates.txt].

WebSocket

HTML5 and WebSocket

The WebSocket protocol was standardized in 2011 with the original goal of allowing browsers to create stable and bidirectional connections with a server. Before that, browsers used to only support HTTPRequests, which is not well-suited for bidirectional communication.

The protocol is message based and a very powerful tool to send push notifications to browsers, and has been used to implement chats, turn-based games, etc. It still uses a TCP connection, which is good for reliability but not for latency, so not good for real-time applications like VoIP and fast-paced games (see WebRTC for those use cases).

Due to its simplicity, its wide compatibility, and being easier to use than a raw TCP connection, WebSocket soon started to spread outside the browsers, in native applications as a mean to communicate with network servers.

Godot supports WebSocket in both native and HTML5 exports.

Using WebSocket in Godot

WebSocket is implemented in Godot via WebSocketPeer. The WebSocket implementation is compatible with the High Level Multiplayer. See section on high-level multiplayer for more details.

Warning

When exporting to Android, make sure to enable the `INTERNET` permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Minimal client example

This example will show you how to create a WebSocket connection to a remote server, and how to send and receive data.

```
extends Node

# The URL we will connect to.
@export var websocket_url = "wss://echo.websocket.org"

# Our WebSocketClient instance.
var socket = WebSocketPeer.new()

func _ready():
    # Initiate connection to the given URL.
    var err = socket.connect_to_url(websocket_url)
    if err != OK:
        print("Unable to connect")
        set_process(false)
    else:
        # Wait for the socket to connect.
        await get_tree().create_timer(2).timeout

        # Send data.
        socket.send_text("Test packet")

func _process(_delta):
    # Call this in _process or _physics_process. Data transfer and state updates
    # will only happen when calling this function.
    socket.poll()

    # get_ready_state() tells you what state the socket is in.
    var state = socket.get_ready_state()
    # WebSocketPeer.STATE_OPEN means the socket is connected
and ready
    # to send and receive data.
    if state == WebSocketPeer.STATE_OPEN:
        while socket.get_available_packet_count():
            print("Got data from server: ",
socket.get_packet().get_string_from_utf8())

    # WebSocketPeer.STATE_CLOSING means the socket is closing.
    # It is important to keep polling for a clean close.
    elif state == WebSocketPeer.STATE_CLOSING:
        pass

    # WebSocketPeer.STATE_CLOSED means the connection has fully
closed.
    # It is now safe to stop polling.
    elif state == WebSocketPeer.STATE_CLOSED:
        # The code will be -1 if the disconnection was not
properly notified by the remote peer.
        var code = socket.get_close_code()
        print("WebSocket closed with code: %d. Clean: %s" %
[code, code != -1])
        set_process(false) # Stop processing.
```

This will print something similar to:

```
Got data from server: Request served by 7811941c69e658
Got data from server: Test packet
```

Minimal server example

This example will show you how to create a WebSocket server that listens for remote connections, and how to send and receive data.

```
extends Node

# The port we will listen to
const PORT = 9080
# Our WebSocketServer instance
```

```
var _server = WebSocketServer.new()

func _ready():
    # Connect base signals to get notified of new client
connections,
    # disconnections, and disconnect requests.
    _server.client_connected.connect(_connected)
    _server.client_disconnected.connect(_disconnected)
    _server.client_close_request.connect(_close_request)
    # This signal is emitted when not using the Multiplayer API
every time a
    # full packet is received.
    # Alternatively, you could check
get_peer(PEER_ID).get_available_packets()
    # in a loop for each connected peer.
    _server.data_received.connect(_on_data)
    # Start listening on the given port.
    var err = _server.listen(PORT)
    if err != OK:
        print("Unable to start server")
        set_process(false)

func _connected(id, proto):
    # This is called when a new peer connects, "id" will be the
assigned peer id,
    # "proto" will be the selected WebSocket sub-protocol
(which is optional)
    print("Client %d connected with protocol: %s" % [id,
proto])

func _close_request(id, code, reason):
    # This is called when a client notifies that it wishes to
close the connection,
    # providing a reason string and close code.
    print("Client %d disconnecting with code: %d, reason: %s" %
[id, code, reason])

func _disconnected(id, was_clean = false):
    # This is called when a client disconnects, "id" will be
the one of the
    # disconnecting client, "was_clean" will tell you if the
disconnection
    # was correctly notified by the remote peer before closing
```

```
the socket.
    print("Client %d disconnected, clean: %s" % [id,
str(was_clean)])

func _on_data(id):
    # Print the received packet, you MUST always use
get_peer(id).get_packet to receive data,
    # and not get_packet directly when not using the
MultiplayerAPI.
    var pkt = _server.get_peer(id).get_packet()
    print("Got data from client %d: %s ... echoing" % [id,
pkt.get_string_from_utf8()])
    _server.get_peer(id).put_packet(pkt)

func _process(delta):
    # Call this in _process or _physics_process.
    # Data transfer, and signals emission will only happen when
calling this function.
    _server.poll()
```

This will print (when a client connects) something similar to this:

```
Client 1348090059 connected with protocol: selected-protocol
Got data from client 1348090059: Test packet ... echoing
```

Advanced chat demo

A more advanced chat demo which optionally uses the multiplayer mid-level abstraction and a high level multiplayer demo are available in the godot demo projects [https://github.com/godotengine/godot-demo-projects] under *networking/websocket_chat* and *networking/websocket_multiplayer*.

WebRTC

HTML5, WebSocket, WebRTC

One of Godot's great features is its ability to export to the HTML5/WebAssembly platform, allowing your game to run directly in the browser when a user visit your webpage.

This is a great opportunity for both demos and full games, but used to come with some limitations. In the area of networking, browsers used to support only HTTPRequests until recently, when first WebSocket and then WebRTC were proposed as standards.

WebSocket

When the WebSocket protocol was standardized in December 2011, it allowed browsers to create stable and bidirectional connections to a WebSocket server. The protocol is a very powerful tool to send push notifications to browsers, and has been used to implement chats, turn-based games, etc.

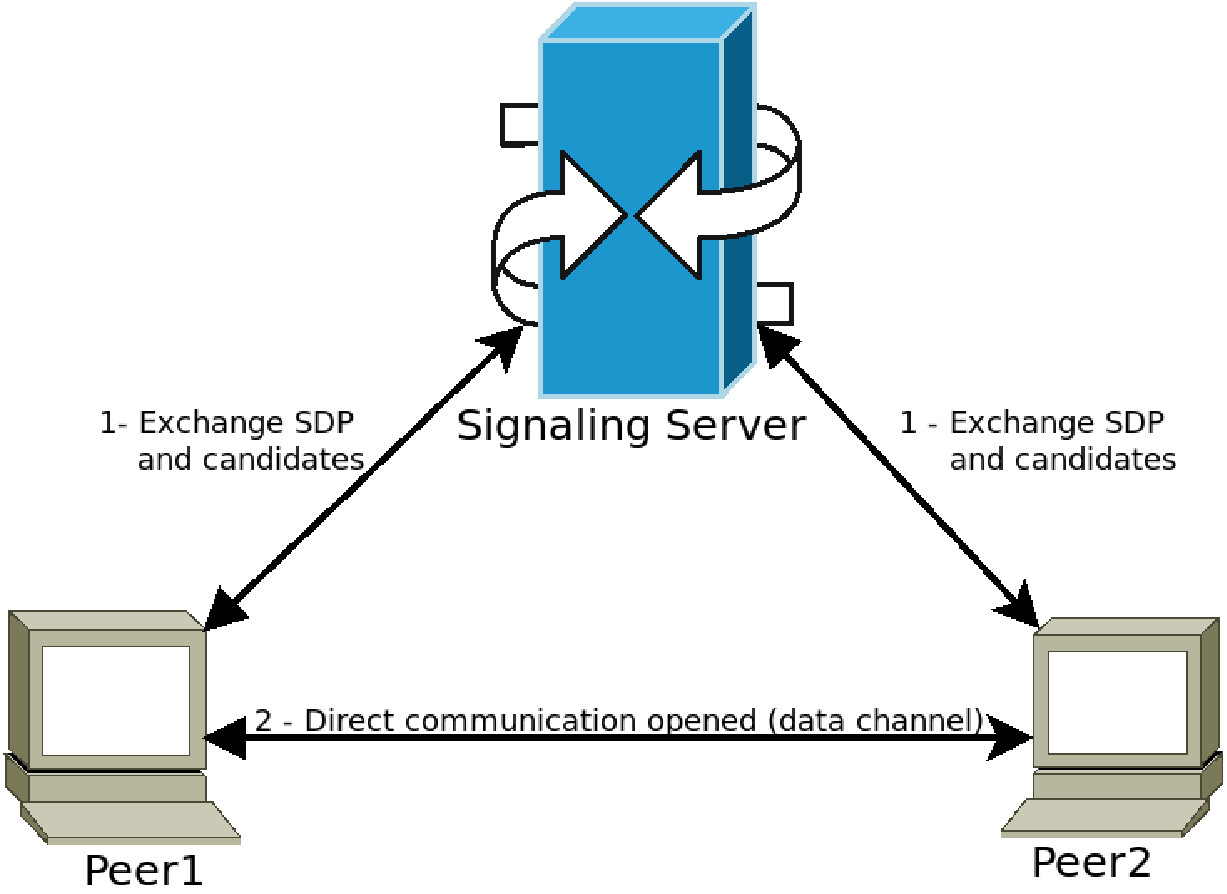
WebSockets, though, still use a TCP connection, which is good for reliability but not for latency, so not good for real-time applications like VoIP and fast-paced games.

WebRTC

For this reason, since 2010, Google started working on a new technology called WebRTC, which later on, in 2017, became a W3C candidate recommendation. WebRTC is a much more complex set of specifications, and relies on many other technologies behind the scenes (ICE, DTLS, SDP) to provide fast, real-time, and secure communication between two peers.

The idea is to find the fastest route between the two peers and establish whenever possible a direct communication (i.e. try to avoid a relaying server).

However, this comes at a price, which is that some media information must be exchanged between the two peers before the communication can start (in the form of Session Description Protocol - SDP strings). This usually takes the form of a so-called WebRTC Signaling Server.


Signaling Server
1- Exchange SDP and candidates
1 - Exchange SDP and candidates
2 - Direct communication opened (data channel)
Peer1
Peer2

Peers connect to a signaling server (for example a WebSocket server) and send their media information. The server then relays this information to other peers, allowing them to establish the desired direct communication. Once this step is done, peers can disconnect from the signaling server and keep the direct Peer-to-Peer (P2P) connection open.

Using WebRTC in Godot

WebRTC is implemented in Godot via two main classes WebRTCPeerConnection and WebRTCDataChannel, plus the multiplayer API implementation WebRTCMultiplayerPeer. See section on high-level multiplayer for more details.

Note

These classes are available automatically in HTML5, but **require an external GDExtension plugin on native (non-HTML5) platforms**. Check out the webrtc-native plugin repository [https://github.com/godotengine/webrtc-native] for instructions and to get the latest release [https://github.com/godotengine/webrtc-native/releases].

Warning

When exporting to Android, make sure to enable the `INTERNET` permission in the Android export preset before exporting the project or using one-click deploy. Otherwise, network communication of any kind will be blocked by Android.

Minimal connection example

This example will show you how to create a WebRTC connection between two peers in the same application. This is not very useful in real life, but will give you a good overview of how a WebRTC connection is set up.

```
extends Node

# Create the two peers
var p1 = WebRTCPeerConnection.new()
var p2 = WebRTCPeerConnection.new()
# And a negotiated channel for each each peer
var ch1 = p1.create_data_channel("chat", {"id": 1,
"negotiated": true})
var ch2 = p2.create_data_channel("chat", {"id": 1,
"negotiated": true})

func _ready():
    # Connect P1 session created to itself to set local
description.

p1.session_description_created.connect(p1.set_local_description
)
    # Connect P1 session and ICE created to p2 set remote
description and candidates.

p1.session_description_created.connect(p2.set_remote_descriptio
n)
    p1.ice_candidate_created.connect(p2.add_ice_candidate)

    # Same for P2

p2.session_description_created.connect(p2.set_local_description
)

p2.session_description_created.connect(p1.set_remote_descriptio
n)
    p2.ice_candidate_created.connect(p1.add_ice_candidate)

    # Let P1 create the offer
    p1.create_offer()

    # Wait a second and send message from P1.
    await get_tree().create_timer(1).timeout
    ch1.put_packet("Hi from P1".to_utf8_buffer())

    # Wait a second and send message from P2.
    await get_tree().create_timer(1).timeout
    ch2.put_packet("Hi from P2".to_utf8_buffer())
```

```
func _process(_delta):
    # Poll connections
    p1.poll()
    p2.poll()

    # Check for messages
    if ch1.get_ready_state() == ch1.STATE_OPEN and
ch1.get_available_packet_count() > 0:
        print("P1 received: ",
ch1.get_packet().get_string_from_utf8())
    if ch2.get_ready_state() == ch2.STATE_OPEN and
ch2.get_available_packet_count() > 0:
        print("P2 received: ",
ch2.get_packet().get_string_from_utf8())
```

This will print:

```
P1 received: Hi from P1
P2 received: Hi from P2
```

Local signaling example

This example expands on the previous one, separating the peers in two different scenes, and using a singleton as a signaling server.

```
extends Node
# An example p2p chat client.

var peer = WebRTCPeerConnection.new()

# Create negotiated data channel.
var channel = peer.create_data_channel("chat", {"negotiated":
true, "id": 1})

func _ready():
    # Connect all functions.
    peer.ice_candidate_created.connect(self._on_ice_candidate)
    peer.session_description_created.connect(self._on_session)
    # Register to the local signaling server (see below for the implementation).
    Signaling.register(String(get_path()))


func _on_ice_candidate(mid, index, sdp):
    # Send the ICE candidate to the other peer via signaling server.
    Signaling.send_candidate(String(get_path()), mid, index, sdp)


func _on_session(type, sdp):
    # Send the session to other peer via signaling server.
    Signaling.send_session(String(get_path()), type, sdp)
    # Set generated description as local.
    peer.set_local_description(type, sdp)


func _process(delta):
    # Always poll the connection frequently.
    peer.poll()
    if channel.get_ready_state() == WebRTCDataChannel.STATE_OPEN:
        while channel.get_available_packet_count() > 0:
            print(String(get_path()), " received: ", channel.get_packet().get_string_from_utf8())


func send_message(message):
    channel.put_packet(message.to_utf8_buffer())
```

And now for the local signaling server:

Note

This local signaling server is supposed to be used as a singleton to connect two peers in the same scene.

```
# A local signaling server. Add this to autoloads with name "Signaling" (/root/Signaling)
extends Node

# We will store the two peers here
var peers = []

func register(path):
    assert(peers.size() < 2)
    peers.append(path)
    if peers.size() == 2:
        get_node(peers[0]).peer.create_offer()


func _find_other(path):
    # Find the other registered peer.
    for p in peers:
        if p != path:
            return p
    return ""


func send_session(path, type, sdp):
    var other = _find_other(path)
    assert(other != "")
    get_node(other).peer.set_remote_description(type, sdp)


func send_candidate(path, mid, index, sdp):
    var other = _find_other(path)
    assert(other != "")
    get_node(other).peer.add_ice_candidate(mid, index, sdp)
```

Then you can use it like this:

```
# Main scene (main.gd)
extends Node

const Chat = preload("res://chat.gd")

func _ready():
    var p1 = Chat.new()
    var p2 = Chat.new()
    add_child(p1)
    add_child(p2)

    # Wait a second and send message from P1
    await get_tree().create_timer(1).timeout
    p1.send_message("Hi from %s" % String(p1.get_path()))

    # Wait a second and send message from P2
    await get_tree().create_timer(1).timeout
    p2.send_message("Hi from %s" % String(p2.get_path()))
```

This will print something similar to this:

```
/root/main/@@3 received: Hi from /root/main/@@2
/root/main/@@2 received: Hi from /root/main/@@3
```

Remote signaling with WebSocket

A more advanced demo using WebSocket for signaling peers and WebRTCMultiplayerPeer is available in the godot demo projects [https://github.com/godotengine/godot-demo-projects] under *networking/webrtc_signaling*.

Performance

Introduction

Godot follows a balanced performance philosophy. In the performance world, there are always tradeoffs, which consist of trading speed for usability and flexibility. Some practical examples of this are:

- Rendering large amounts of objects efficiently is easy, but when a large scene must be rendered, it can become inefficient. To solve this, visibility computation must be added to the rendering. This makes rendering less efficient, but at the same time, fewer objects are rendered. Therefore, the overall rendering efficiency is improved.
- Configuring the properties of every material for every object that needs to be rendered is also slow. To solve this, objects are sorted by material to reduce the costs. At the same time, sorting has a cost.
- In 3D physics, a similar situation happens. The best algorithms to handle large amounts of physics objects (such as SAP) are slow at insertion/removal of objects and raycasting. Algorithms that allow faster insertion and removal, as well as raycasting, will not be able to handle as many active objects.

And there are many more examples of this! Game engines strive to be general-purpose in nature. Balanced algorithms are always favored over algorithms that might be fast in some situations and slow in others, or algorithms that are fast but are more difficult to use.

Godot is not an exception to this. While it is designed to have backends swappable for different algorithms, the default backends prioritize balance and flexibility over performance.

With this clear, the aim of this tutorial section is to explain how to get the maximum performance out of Godot. While the tutorials can be read in any order, it is a good idea to start from General optimization tips.

Common

- General optimization tips
- Optimization using Servers

CPU

- CPU optimization

GPU

- GPU optimization
- Optimization using MultiMeshes

3D

- Optimizing 3D performance
- Animating thousands of objects

Threads

- Using multiple threads
- Thread-safe APIs

General optimization tips

Introduction

In an ideal world, computers would run at infinite speed. The only limit to what we could achieve would be our imagination. However, in the real world, it's all too easy to produce software that will bring even the fastest computer to its knees.

Thus, designing games and other software is a compromise between what we would like to be possible, and what we can realistically achieve while maintaining good performance.

To achieve the best results, we have two approaches:

- Work faster.
- Work smarter.

And preferably, we will use a blend of the two.

Smoke and mirrors

Part of working smarter is recognizing that, in games, we can often get the player to believe they're in a world that is far more complex, interactive, and graphically exciting than it really is. A good programmer is a magician, and should strive to learn the tricks of the trade while trying to invent new ones.

The nature of slowness

To the outside observer, performance problems are often lumped together. But in reality, there are several different kinds of performance problems:

- A slow process that occurs every frame, leading to a continuously low frame rate.
- An intermittent process that causes "spikes" of slowness, leading to stalls.
- A slow process that occurs outside of normal gameplay, for instance, when loading a level.

Each of these are annoying to the user, but in different ways.

Measuring performance

Probably the most important tool for optimization is the ability to measure performance - to identify where bottlenecks are, and to measure the success of our attempts to speed them up.

There are several methods of measuring performance, including:

- Putting a start/stop timer around code of interest.
- Using the Godot profiler.
- Using external CPU profilers.
- Using external GPU profilers/debuggers such as NVIDIA Nsight Graphics [https://developer.nvidia.com/nsight-graphics], Radeon GPU Profiler [https://gpuopen.com/rgp/], Intel Graphics Performance Analyzers [https://www.intel.com/content/www/us/en/developer/tools/graphics-performance-analyzers/overview.html], or Arm Performance Studio [https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Studio].
- Checking the frame rate (with V-Sync disabled). Third-party utilities such as RivaTuner Statistics Server [https://www.guru3d.com/files-details/rtss-rivatuner-statistics-server-download.html] (Windows) or MangoHud

[https://github.com/flightlessmango/MangoHud] (Linux) can also be useful here.
- Using an unofficial debug menu add-on [https://github.com/godot-extended-libraries/godot-debug-menu].

Be very aware that the relative performance of different areas can vary on different hardware. It's often a good idea to measure timings on more than one device. This is especially the case if you're targeting mobile devices.

Limitations

CPU profilers are often the go-to method for measuring performance. However, they don't always tell the whole story.

- Bottlenecks are often on the GPU, "as a result" of instructions given by the CPU.
- Spikes can occur in the operating system processes (outside of Godot) "as a result" of instructions used in Godot (for example, dynamic memory allocation).
- You may not always be able to profile specific devices like a mobile phone due to the initial setup required.
- You may have to solve performance problems that occur on hardware you don't have access to.

As a result of these limitations, you often need to use detective work to find out where bottlenecks are.

Detective work

Detective work is a crucial skill for developers (both in terms of performance, and also in terms of bug fixing). This can include hypothesis testing, and binary search.

Hypothesis testing

Say, for example, that you believe sprites are slowing down your game. You can test this hypothesis by:

- Measuring the performance when you add more sprites, or take some away.

This may lead to a further hypothesis: does the size of the sprite determine the performance drop?

- You can test this by keeping everything the same, but changing the sprite size, and measuring performance.

Binary search

If you know that frames are taking much longer than they should, but you're not sure where the bottleneck lies. You could begin by commenting out approximately half the routines that occur on a normal frame. Has the performance improved more or less than expected?

Once you know which of the two halves contains the bottleneck, you can repeat this process until you've pinned down the problematic area.

Profilers

Profilers allow you to time your program while running it. Profilers then provide results telling you what percentage of time was spent in different functions and areas, and how often functions were called.

This can be very useful both to identify bottlenecks and to measure the results of your improvements. Sometimes, attempts to improve performance can backfire and lead to slower performance. **Always use profiling and timing to guide your efforts.**

For more info about using Godot's built-in profiler, see The Profiler.

Principles

Donald Knuth [https://en.wikipedia.org/wiki/Donald_Knuth] said:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

The messages are very important:

- Developer time is limited. Instead of blindly trying to speed up all aspects of a program, we should concentrate our efforts on the aspects that really matter.
- Efforts at optimization often end up with code that is harder to read and debug than non-optimized code. It is in our interests to limit this to areas that will really benefit.

Just because we *can* optimize a particular bit of code, it doesn't necessarily mean that we *should*. Knowing when and when not to optimize is a great skill to develop.

One misleading aspect of the quote is that people tend to focus on the subquote *"premature optimization is the root of all evil"*. While *premature* optimization is (by definition) undesirable, performant software is the result of performant design.

Performant design

The danger with encouraging people to ignore optimization until necessary, is that it conveniently ignores that the most important time to consider performance is at the design stage, before a key has even hit a keyboard. If the design or algorithms of a program are inefficient, then no amount of polishing the details later will make it run fast. It may run *faster*, but it will never run as fast as a program designed for performance.

This tends to be far more important in game or graphics programming than in general programming. A performant design, even without low-level optimization, will often run many times faster than a mediocre design with low-level optimization.

Incremental design

Of course, in practice, unless you have prior knowledge, you are unlikely to come up with the best design the first time. Instead, you'll often make a series of versions of a particular area of code, each taking a different approach to the problem, until you come to a satisfactory solution. It's important not to spend too much time on the details at this stage until you have finalized the overall design. Otherwise, much of your work will be thrown out.

It's difficult to give general guidelines for performant design because this is so dependent on the problem. One point worth mentioning though, on the CPU side, is that modern CPUs are nearly always limited by memory bandwidth. This has led to a resurgence in data-oriented design, which involves designing data structures and algorithms for *cache locality* of data and linear access, rather than jumping around in memory.

The optimization process

Assuming we have a reasonable design, and taking our lessons from Knuth, our first step in optimization should be to identify the biggest bottlenecks - the slowest functions, the low-hanging fruit.

Once we've successfully improved the speed of the slowest area, it may no longer be the bottleneck. So we should test/profile again and find the next bottleneck on which to focus.

The process is thus:

1. Profile / Identify bottleneck.
2. Optimize bottleneck.
3. Return to step 1.

Optimizing bottlenecks

Some profilers will even tell you which part of a function (which data accesses, calculations) are slowing things down.

As with design, you should concentrate your efforts first on making sure the algorithms and data structures are the best they can be. Data access should be local (to make best use of CPU cache), and it can often be better to use compact storage of data (again, always profile to test results). Often, you precalculate heavy computations ahead of time. This can be done by performing the computation when loading a level, by loading a file containing precalculated data or simply by storing the results of complex calculations into a script constant and reading its value.

Once algorithms and data are good, you can often make small changes in routines which improve performance. For instance, you can move some calculations outside of loops or transform nested `for` loops into non-nested loops. (This should be feasible if you know a 2D array's width or height in advance.)

Always retest your timing/bottlenecks after making each change. Some changes will increase speed, others may have a negative effect. Sometimes, a small positive effect will be outweighed by the negatives of more complex code, and you may choose to leave out that optimization.

Appendix

Bottleneck math

The proverb *"a chain is only as strong as its weakest link"* applies directly to performance optimization. If your project is spending 90% of the time in function `A`, then optimizing `A` can have a massive effect on performance.

```
A: 9 ms
Everything else: 1 ms
Total frame time: 10 ms
```

```
A: 1 ms
Everything else: 1ms
Total frame time: 2 ms
```

In this example, improving this bottleneck `A` by a factor of 9× decreases overall frame time by 5× while increasing frames per second by 5×.

However, if something else is running slowly and also bottlenecking your project, then the same improvement can lead to less dramatic gains:

```
A: 9 ms
Everything else: 50 ms
Total frame time: 59 ms
```

```
A: 1 ms
Everything else: 50 ms
```

```
Total frame time: 51 ms
```

In this example, even though we have hugely optimized function `A`, the actual gain in terms of frame rate is quite small.

In games, things become even more complicated because the CPU and GPU run independently of one another. Your total frame time is determined by the slower of the two.

```
CPU: 9 ms
GPU: 50 ms
Total frame time: 50 ms
```

```
CPU: 1 ms
GPU: 50 ms
Total frame time: 50 ms
```

In this example, we optimized the CPU hugely again, but the frame time didn't improve because we are GPU-bottlenecked.

Optimization using Servers

Engines like Godot provide increased ease of use thanks to their high level constructs and features. Most of them are accessed and used via the Scene System. Using nodes and resources simplifies project organization and asset management in complex games.

There are, of course, always drawbacks:

- There is an extra layer of complexity.
- Performance is lower than when using simple APIs directly.
- It is not possible to use multiple threads to control them.
- More memory is needed.

In many cases, this is not really a problem (Godot is very optimized, and most operations are handled with signals, so no polling is required). Still, sometimes it can be. For example, dealing with tens of thousands of instances for something that needs to be processed every frame can be a bottleneck.

This type of situation makes programmers regret they are using a game engine and wish they could go back to a more handcrafted, low level implementation of game code.

Still, Godot is designed to work around this problem.

See also

You can see how using low-level servers works in action using the Bullet Shower demo project [https://github.com/godotengine/godot-demo-projects/tree/master/2d/bullet_shower]

Servers

One of the most interesting design decisions for Godot is the fact that the whole scene system is *optional*. While it is not currently possible to compile it out, it can be completely bypassed.

At the core, Godot uses the concept of Servers. They are very low-level APIs to control rendering, physics, sound, etc. The scene system is built on top of them and uses them directly. The most common servers are:

- RenderingServer: handles everything related to graphics.
- PhysicsServer3D: handles everything related to 3D physics.
- PhysicsServer2D: handles everything related to 2D physics.
- AudioServer: handles everything related to audio.

Explore their APIs and you will realize that all the functions provided are low-level implementations of everything Godot allows you to do.

RIDs

The key to using servers is understanding Resource ID (RID) objects. These are opaque handles to the server implementation. They are allocated and freed manually. Almost every function in the servers requires RIDs to access the actual resource.

Most Godot nodes and resources contain these RIDs from the servers internally, and they can be obtained with different functions. In fact, anything that inherits Resource can be directly casted to an RID. Not all resources contain an RID, though: in such cases, the RID will be empty. The resource can then be passed to server APIs as an RID.

Warning

> Resources are reference-counted (see RefCounted), and references to a resource's RID are *not* counted when determining whether the resource is still in use. Make sure to keep a reference to the resource outside the server, or else both it and its RID will be erased.

For nodes, there are many functions available:

- For CanvasItem, the CanvasItem.get_canvas_item() method will return the canvas item RID in the server.
- For CanvasLayer, the CanvasLayer.get_canvas() method will return the canvas RID in the server.
- For Viewport, the Viewport.get_viewport_rid() method will return the viewport RID in the server.
- For 3D, the World3D resource (obtainable in the Viewport and Node3D nodes) contains functions to get the *RenderingServer Scenario*, and the *PhysicsServer Space*. This allows creating 3D objects directly with the server API and using them.
- For 2D, the World2D resource (obtainable in the Viewport and CanvasItem nodes) contains functions to get the *RenderingServer Canvas*, and the *Physics2DServer Space*. This allows creating 2D objects directly with the server API and using them.
- The VisualInstance3D class, allows getting the scenario *instance* and *instance base* via the VisualInstance3D.get_instance() and VisualInstance3D.get_base() respectively.

Try exploring the nodes and resources you are familiar with and find the functions to obtain the server *RIDs*.

It is not advised to control RIDs from objects that already have a node associated. Instead, server functions should always be used for creating and controlling new ones and interacting with the existing ones.

Creating a sprite

This is an example of how to create a sprite from code and move it using the low-level CanvasItem API.

GDScript

```
extends Node2D


# RenderingServer expects references to be kept around.
var texture


func _ready():
    # Create a canvas item, child of this node.
    var ci_rid = RenderingServer.canvas_item_create()
    # Make this node the parent.
    RenderingServer.canvas_item_set_parent(ci_rid, get_canvas_item())
    # Draw a texture on it.
    # Remember, keep this reference.
    texture = load("res://my_texture.png")
    # Add it, centered.
    RenderingServer.canvas_item_add_texture_rect(ci_rid, Rect2(-texture.get_size() / 2, texture.get_size()), texture)
    # Add the item, rotated 45 degrees and translated.
    var xform = Transform2D().rotated(deg_to_rad(45)).translated(Vector2(20, 30))
    RenderingServer.canvas_item_set_transform(ci_rid, xform)
```

C#

```
public partial class MyNode2D : Node2D
{
    // RenderingServer expects references to be kept around.
    private Texture2D _texture;

    public override void _Ready()
    {
        // Create a canvas item, child of this node.
        Rid ciRid = RenderingServer.CanvasItemCreate();
        // Make this node the parent.
        RenderingServer.CanvasItemSetParent(ciRid,
GetCanvasItem());
        // Draw a texture on it.
        // Remember, keep this reference.
        _texture = ResourceLoader.Load<Texture2D>
("res://MyTexture.png");
        // Add it, centered.
        RenderingServer.CanvasItemAddTextureRect(ciRid, new
Rect2(-_texture.GetSize() / 2, _texture.GetSize()),
_texture.GetRid());
        // Add the item, rotated 45 degrees and translated.
        Transform2D xform =
Transform2D.Identity.Rotated(Mathf.DegToRad(45)).Translated(new
Vector2(20, 30));
        RenderingServer.CanvasItemSetTransform(ciRid, xform);
    }
}
```

The Canvas Item API in the server allows you to add draw primitives to it. Once added, they can't be modified. The Item needs to be cleared and the primitives re-added (this is not the case for setting the transform, which can be done as many times as desired).

Primitives are cleared this way:

GDScript

```
RenderingServer.canvas_item_clear(ci_rid)
```

C#

```
RenderingServer.CanvasItemClear(ciRid);
```

Instantiating a Mesh into 3D space

The 3D APIs are different from the 2D ones, so the instantiation API must be used.

GDScript

```
extends Node3D


# RenderingServer expects references to be kept around.
var mesh


func _ready():
    # Create a visual instance (for 3D).
    var instance = RenderingServer.instance_create()
    # Set the scenario from the world, this ensures it
    # appears with the same objects as the scene.
    var scenario = get_world_3d().scenario
    RenderingServer.instance_set_scenario(instance, scenario)
    # Add a mesh to it.
    # Remember, keep the reference.
    mesh = load("res://mymesh.obj")
    RenderingServer.instance_set_base(instance, mesh)
    # Move the mesh around.
    var xform = Transform3D(Basis(), Vector3(20, 100, 0))
    RenderingServer.instance_set_transform(instance, xform)
```

C#

```
public partial class MyNode3D : Node3D
{
    // RenderingServer expects references to be kept around.
    private Mesh _mesh;

    public override void _Ready()
    {
        // Create a visual instance (for 3D).
        Rid instance = RenderingServer.InstanceCreate();
        // Set the scenario from the world, this ensures it
        // appears with the same objects as the scene.
        Rid scenario = GetWorld3D().Scenario;
        RenderingServer.InstanceSetScenario(instance,
scenario);
        // Add a mesh to it.
        // Remember, keep the reference.
        _mesh = ResourceLoader.Load<Mesh>("res://MyMesh.obj");
        RenderingServer.InstanceSetBase(instance,
_mesh.GetRid());
        // Move the mesh around.
        Transform3D xform = new Transform3D(Basis.Identity, new
Vector3(20, 100, 0));
        RenderingServer.InstanceSetTransform(instance, xform);
    }
}
```

Creating a 2D RigidBody and moving a sprite with it

This creates a RigidBody2D using the PhysicsServer2D API, and moves a CanvasItem when the body moves.

GDScript

```
# Physics2DServer expects references to be kept around.
var body
var shape


func _body_moved(state, index):
    # Created your own canvas item, use it here.
    RenderingServer.canvas_item_set_transform(canvas_item,
state.transform)


func _ready():
    # Create the body.
    body = Physics2DServer.body_create()
    Physics2DServer.body_set_mode(body,
Physics2DServer.BODY_MODE_RIGID)
    # Add a shape.
    shape = Physics2DServer.rectangle_shape_create()
    # Set rectangle extents.
    Physics2DServer.shape_set_data(shape, Vector2(10, 10))
    # Make sure to keep the shape reference!
    Physics2DServer.body_add_shape(body, shape)
    # Set space, so it collides in the same space as current
scene.
    Physics2DServer.body_set_space(body, get_world_2d().space)
    # Move initial position.
    Physics2DServer.body_set_state(body,
Physics2DServer.BODY_STATE_TRANSFORM, Transform2D(0,
Vector2(10, 20)))
    # Add the transform callback, when body moves
    # The last parameter is optional, can be used as index
    # if you have many bodies and a single callback.
    Physics2DServer.body_set_force_integration_callback(body,
self, "_body_moved", 0)
```

The 3D version should be very similar, as 2D and 3D physics servers are identical (using RigidBody3D and PhysicsServer3D respectively).

Getting data from the servers

Try to **never** request any information from `RenderingServer`, `PhysicsServer2D` or `PhysicsServer3D` by calling functions unless you know what you are doing. These servers will often run asynchronously for performance and calling any function that returns a value will stall them and force them to process anything pending until the function is actually called. This will severely decrease performance if you call them every frame (and it won't be obvious why).

Because of this, most APIs in such servers are designed so it's not even possible to request information back, until it's actual data that can be saved.

CPU optimization

Measuring performance

We have to know where the "bottlenecks" are to know how to speed up our program. Bottlenecks are the slowest parts of the program that limit the rate that everything can progress. Focusing on bottlenecks allows us to concentrate our efforts on optimizing the areas which will give us the greatest speed improvement, instead of spending a lot of time optimizing functions that will lead to small performance improvements.

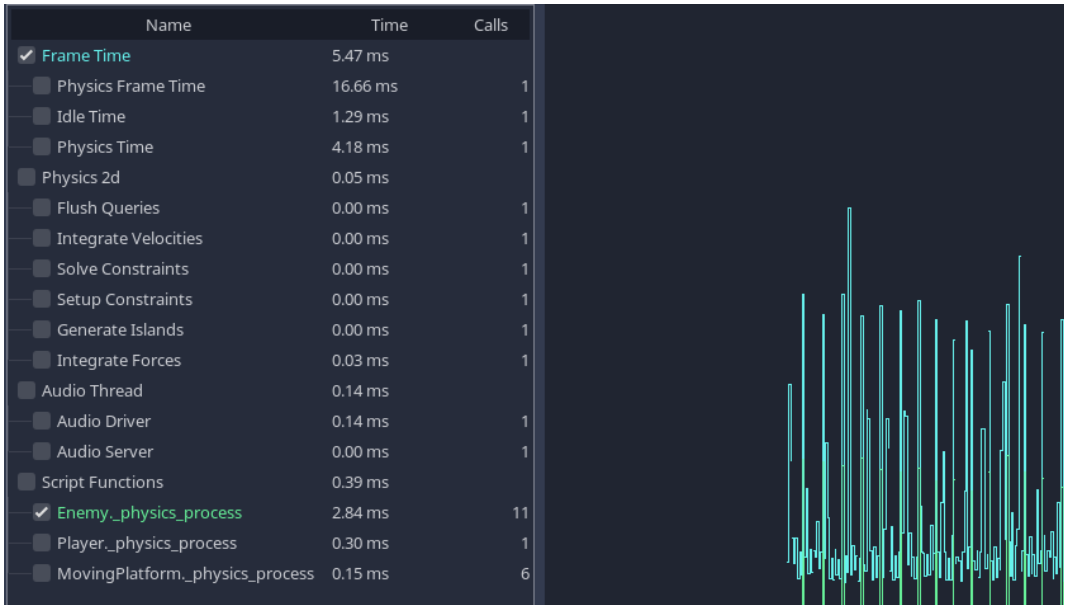
For the CPU, the easiest way to identify bottlenecks is to use a profiler.

CPU profilers

Profilers run alongside your program and take timing measurements to work out what proportion of time is spent in each function.

The Godot IDE conveniently has a built-in profiler. It does not run every time you start your project: it must be manually started and stopped. This is because, like most profilers, recording these timing measurements can slow down your project significantly.

After profiling, you can look back at the results for a frame.



Name	Time	Calls
☑ Frame Time	5.47 ms	
☐ Physics Frame Time	16.66 ms	1
☐ Idle Time	1.29 ms	1
☐ Physics Time	4.18 ms	1
☐ Physics 2d	0.05 ms	
☐ Flush Queries	0.00 ms	1
☐ Integrate Velocities	0.00 ms	1
☐ Solve Constraints	0.00 ms	1
☐ Setup Constraints	0.00 ms	1
☐ Generate Islands	0.00 ms	1
☐ Integrate Forces	0.03 ms	1
☐ Audio Thread	0.14 ms	
☐ Audio Driver	0.14 ms	1
☐ Audio Server	0.00 ms	1
☐ Script Functions	0.39 ms	
☑ Enemy._physics_process	2.84 ms	11
☐ Player._physics_process	0.30 ms	1
☐ MovingPlatform._physics_process	0.15 ms	6

Results of a profile of one of the demo projects.

Note

We can see the cost of built-in processes such as physics and audio, as well as seeing the cost of our own scripting functions at the bottom.

Time spent waiting for various built-in servers may not be counted in the profilers. This is a known bug.

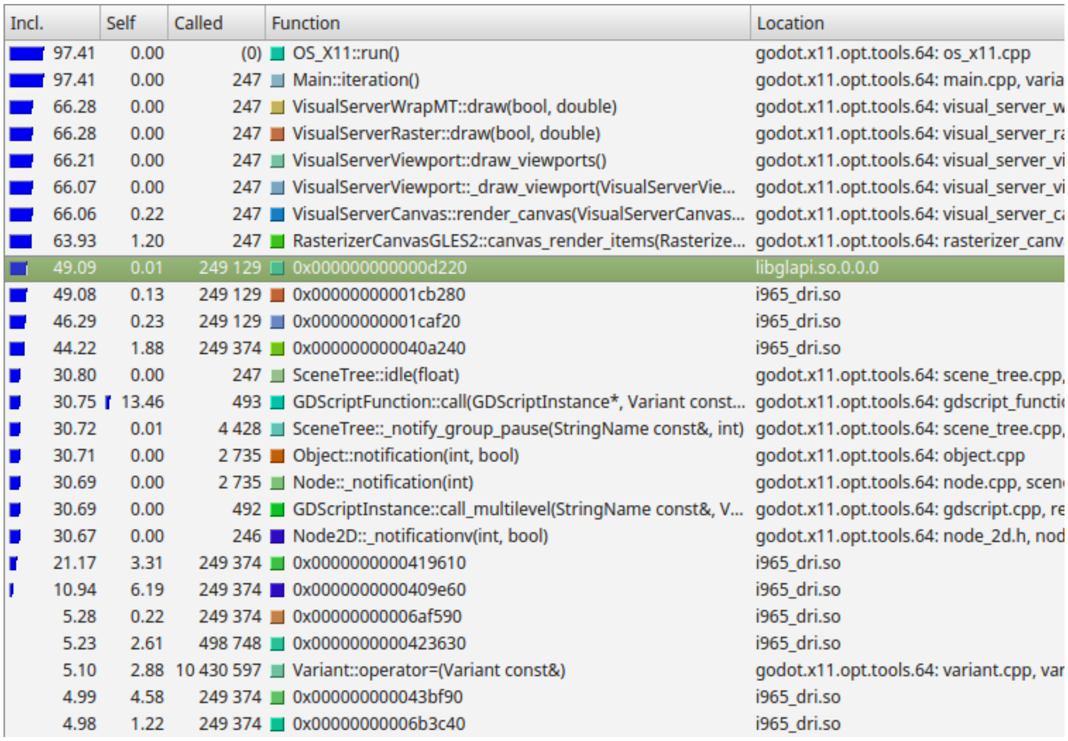
When a project is running slowly, you will often see an obvious function or process taking a lot more time than others. This is your primary bottleneck, and you can usually increase speed by optimizing this area.

For more info about using Godot's built-in profiler, see Debugger panel.

External profilers

Although the Godot IDE profiler is very convenient and useful, sometimes you need more power, and the ability to profile the Godot engine source code itself.

You can use a number of third-party C++ profilers to do this.



Incl.	Self	Called	Function	Location
97.41	0.00	(0)	OS_X11::run()	godot.x11.opt.tools.64: os_x11.cpp
97.41	0.00	247	Main::iteration()	godot.x11.opt.tools.64: main.cpp, varia
66.28	0.00	247	VisualServerWrapMT::draw(bool, double)	godot.x11.opt.tools.64: visual_server_w
66.28	0.00	247	VisualServerRaster::draw(bool, double)	godot.x11.opt.tools.64: visual_server_ra
66.21	0.00	247	VisualServerViewport::draw_viewports()	godot.x11.opt.tools.64: visual_server_vi
66.07	0.00	247	VisualServerViewport::_draw_viewport(VisualServerVie...	godot.x11.opt.tools.64: visual_server_vi
66.06	0.22	247	VisualServerCanvas::render_canvas(VisualServerCanvas...	godot.x11.opt.tools.64: visual_server_ca
63.93	1.20	247	RasterizerCanvasGLES2::canvas_render_items(Rasterize...	godot.x11.opt.tools.64: rasterizer_canv
49.09	0.01	249 129	0x000000000000d220	libglapi.so.0.0.0
49.08	0.13	249 129	0x00000000001cb280	i965_dri.so
46.29	0.23	249 129	0x00000000001caf20	i965_dri.so
44.22	1.88	249 374	0x000000000040a240	i965_dri.so
30.80	0.00	247	SceneTree::idle(float)	godot.x11.opt.tools.64: scene_tree.cpp,
30.75	13.46	493	GDScriptFunction::call(GDScriptInstance*, Variant const...	godot.x11.opt.tools.64: gdscript_functi
30.72	0.01	4 428	SceneTree::_notify_group_pause(StringName const&, int)	godot.x11.opt.tools.64: scene_tree.cpp,
30.71	0.00	2 735	Object::notification(int, bool)	godot.x11.opt.tools.64: object.cpp
30.69	0.00	2 735	Node::_notification(int)	godot.x11.opt.tools.64: node.cpp, scen
30.69	0.00	492	GDScriptInstance::call_multilevel(StringName const&, V...	godot.x11.opt.tools.64: gdscript.cpp, re
30.67	0.00	246	Node2D::_notificationv(int, bool)	godot.x11.opt.tools.64: node_2d.h, nod
21.17	3.31	249 374	0x0000000000419610	i965_dri.so
10.94	6.19	249 374	0x0000000000409e60	i965_dri.so
5.28	0.22	249 374	0x00000000006af590	i965_dri.so
5.23	2.61	498 748	0x0000000000423630	i965_dri.so
5.10	2.88	10 430 597	Variant::operator=(Variant const&)	godot.x11.opt.tools.64: variant.cpp, var
4.99	4.58	249 374	0x000000000043bf90	i965_dri.so
4.98	1.22	249 374	0x00000000006b3c40	i965_dri.so

Example results from Callgrind, which is part of Valgrind.

From the left, Callgrind is listing the percentage of time within a function and its children (Inclusive), the percentage of time spent within the function itself, excluding child functions (Self), the number of times the function is called, the function name, and the file or module.

In this example, we can see nearly all time is spent under the `Main::iteration()` function. This is the master function in the Godot source code that is called repeatedly. It causes frames to be drawn, physics ticks to be simulated, and nodes and scripts to be updated. A large proportion of the time is spent in the functions to render a canvas (66%), because this example uses a 2D benchmark. Below this, we see that almost 50% of the time is spent outside Godot code in `libglapi` and `i965_dri` (the graphics driver). This tells us the a large proportion of CPU time is being spent in the graphics driver.

This is actually an excellent example because, in an ideal world, only a very small proportion of time would be spent in the graphics driver. This is an indication that there is a problem with too much communication and work being done in the graphics API. This specific profiling led to the development of 2D batching, which greatly speeds up 2D rendering by reducing bottlenecks in this area.

Manually timing functions

Another handy technique, especially once you have identified the bottleneck using a profiler, is to manually time the function or area under test. The specifics vary depending on the language, but in GDScript, you would do the following:

GDScript

```
var time_start = Time.get_ticks_usec()

# Your function you want to time
update_enemies()

var time_end = Time.get_ticks_usec()
print("update_enemies() took %d microseconds" % time_end -
time_start)
```

C#

```
var timeStart = Time.GetTicksUsec();

// Your function you want to time.
UpdateEnemies();

var timeEnd = Time.GetTicksUsec();
```

```
GD.Print($"UpdateEnemies() took {timeEnd - timeStart}
microseconds");
```

When manually timing functions, it is usually a good idea to run the function many times (1,000 or more times), instead of just once (unless it is a very slow function). The reason for doing this is that timers often have limited accuracy. Moreover, CPUs will schedule processes in a haphazard manner. Therefore, an average over a series of runs is more accurate than a single measurement.

As you attempt to optimize functions, be sure to either repeatedly profile or time them as you go. This will give you crucial feedback as to whether the optimization is working (or not).

Caches

CPU caches are something else to be particularly aware of, especially when comparing timing results of two different versions of a function. The results can be highly dependent on whether the data is in the CPU cache or not. CPUs don't load data directly from the system RAM, even though it's huge in comparison to the CPU cache (several gigabytes instead of a few megabytes). This is because system RAM is very slow to access. Instead, CPUs load data from a smaller, faster bank of memory called cache. Loading data from cache is very fast, but every time you try and load a memory address that is not stored in cache, the cache must make a trip to main memory and slowly load in some data. This delay can result in the CPU sitting around idle for a long time, and is referred to as a "cache miss".

This means that the first time you run a function, it may run slowly because the data is not in the CPU cache. The second and later times, it may run much faster because the data is in the cache. Due to this, always use averages when timing, and be aware of the effects of cache.

Understanding caching is also crucial to CPU optimization. If you have an algorithm (routine) that loads small bits of data from randomly spread out areas of main memory, this can result in a lot of cache misses, a lot of the time, the CPU will be waiting around for data instead of doing any work. Instead, if you can make your data accesses localised, or even better, access memory in a linear fashion (like a continuous list), then the cache will work optimally and the CPU will be able to work as fast as possible.

Godot usually takes care of such low-level details for you. For example, the Server APIs make sure data is optimized for caching already for things like rendering and physics. Still, you should be especially aware of caching when writing GDExtensions.

Languages

Godot supports a number of different languages, and it is worth bearing in mind that there are trade-offs involved. Some languages are designed for ease of use at the cost of speed, and others are faster but more difficult to work with.

Built-in engine functions run at the same speed regardless of the scripting language you choose. If your project is making a lot of calculations in its own code, consider moving those calculations to a faster language.

GDScript

GDScript is designed to be easy to use and iterate, and is ideal for making many types of games. However, in this language, ease of use is considered more important than performance. If you need to make heavy calculations, consider moving some of your project to one of the other languages.

C#

C# is popular and has first-class support in Godot. It offers a good compromise between speed and ease of use. Beware of possible garbage collection pauses and leaks that can occur during gameplay, though. A common approach to workaround issues with garbage collection is to use *object pooling*, which is outside the scope of this guide.

Other languages

Third parties provide support for several other languages, including Rust [https://github.com/godot-rust/gdext].

C++

Godot is written in C++. Using C++ will usually result in the fastest code. However, on a practical level, it is the most difficult to deploy to end users' machines on different platforms. Options for using C++ include GDExtensions and custom modules.

Threads

Consider using threads when making a lot of calculations that can run in parallel to each other. Modern CPUs have multiple cores, each one capable of doing a limited amount of work. By spreading work over multiple threads, you can move further towards peak CPU efficiency.

The disadvantage of threads is that you have to be incredibly careful. As each CPU core operates independently, they can end up trying to access the same memory at the same time. One thread can be reading to a variable while another is writing: this is called a *race* *condition*. Before you use threads, make sure you understand the dangers and how to try and prevent these race conditions. Threads can make debugging considerably more difficult.

For more information on threads, see Using multiple threads.

SceneTree

Although Nodes are an incredibly powerful and versatile concept, be aware that every node has a cost. Built-in functions such as `_process()` and `_physics_process()` propagate through the tree. This housekeeping can reduce performance when you have a very large numbers of nodes (how many exactly depends on the target platform and can range from thousands to tens of thousands so ensure that you profile performance on all target platforms during development).

Each node is handled individually in the Godot renderer. Therefore, a smaller number of nodes with more in each can lead to better performance.

One quirk of the SceneTree is that you can sometimes get much better performance by removing nodes from the SceneTree, rather than by pausing or hiding them. You don't have to delete a detached node. You can for example, keep a reference to a node, detach it from the scene tree using Node.remove_child(node), then reattach it later using Node.add_child(node). This can be very useful for adding and removing areas from a game, for example.

You can avoid the SceneTree altogether by using Server APIs. For more information, see Optimization using Servers.

Physics

In some situations, physics can end up becoming a bottleneck. This is particularly the case with complex worlds and large numbers of physics objects.

Here are some techniques to speed up physics:

- Try using simplified versions of your rendered geometry for collision shapes. Often, this won't be noticeable for end users, but can greatly increase performance.
- Try removing objects from physics when they are out of view / outside the current area, or reusing physics objects (maybe you allow 8 monsters per area, for example, and reuse these).

Another crucial aspect to physics is the physics tick rate. In some games, you can greatly reduce the tick rate, and instead of for example, updating physics 60 times per second, you may update them only 30 or even 20 times per second. This can greatly reduce the CPU load.

The downside of changing physics tick rate is you can get jerky movement or jitter when the physics update rate does not match the frames per second rendered. Also, decreasing the physics tick rate will increase input lag. It's recommended to stick to the default physics tick rate (60 Hz) in most games that feature real-time player movement.

The solution to jitter is to use *fixed timestep interpolation*, which involves smoothing the rendered positions and rotations over multiple frames to match the physics. You can either implement this yourself or use a third-party addon [https://github.com/lawnjelly/smoothing-addon]. Performance-wise, interpolation is a very cheap operation compared to running a physics tick. It's orders of magnitude faster, so this can be a significant performance win while also reducing jitter.

GPU optimization

Introduction

The demand for new graphics features and progress almost guarantees that you will encounter graphics bottlenecks. Some of these can be on the CPU side, for instance in calculations inside the Godot engine to prepare objects for rendering. Bottlenecks can also occur on the CPU in the graphics driver, which sorts instructions to pass to the GPU, and in the transfer of these instructions. And finally, bottlenecks also occur on the GPU itself.

Where bottlenecks occur in rendering is highly hardware-specific. Mobile GPUs in particular may struggle with scenes that run easily on desktop.

Understanding and investigating GPU bottlenecks is slightly different to the situation on the CPU. This is because, often, you can only change performance indirectly by changing the instructions you give to the GPU. Also, it may be more difficult to take measurements. In many cases, the only way of measuring performance is by examining changes in the time spent rendering each frame.

Draw calls, state changes, and APIs

Note

The following section is not relevant to end-users, but is useful to provide background information that is relevant in later sections.

Godot sends instructions to the GPU via a graphics API (Vulkan, OpenGL, OpenGL ES or WebGL). The communication and driver activity involved can be quite costly, especially in OpenGL, OpenGL ES and WebGL. If we can provide these instructions in a way that is preferred by the driver and GPU, we can greatly increase performance.

Nearly every API command in OpenGL requires a certain amount of validation to make sure the GPU is in the correct state. Even seemingly simple commands can lead to a flurry of behind-the-scenes housekeeping. Therefore, the goal is to reduce these instructions to a bare minimum and group together similar objects as much as possible so they can be rendered together, or with the minimum number of these expensive state changes.

2D batching

In 2D, the costs of treating each item individually can be prohibitively high - there can easily be thousands of them on the screen. This is why 2D *batching* is used with OpenGL-based rendering methods. Multiple similar items are grouped together and rendered in a batch, via a single draw call, rather than making a separate draw call for each item. In addition, this means state changes, material and texture changes can be kept to a minimum.

Vulkan-based rendering methods do not use 2D batching yet. Since draw calls are much cheaper with Vulkan compared to OpenGL, there is less of a need to have 2D batching (although it can still be beneficial in some cases).

3D batching

In 3D, we still aim to minimize draw calls and state changes. However, it can be more difficult to batch together several objects into a single draw call. 3D meshes tend to comprise hundreds or thousands of triangles, and combining large meshes in real-time is prohibitively expensive. The costs of joining them quickly exceeds any benefits as the number of triangles grows per mesh. A much better alternative is to **join meshes ahead of time** (static meshes in relation to each other). This can be done by artists, or programmatically within Godot using an add-on.

There is also a cost to batching together objects in 3D. Several objects rendered as one cannot be individually culled. An entire city that is off-screen will still be rendered if it is joined to a single blade of grass that is on screen. Thus, you should always take objects' location and culling into account when attempting to batch 3D objects together. Despite this, the benefits of joining static objects often outweigh other considerations, especially for large numbers of distant or low-poly objects.

For more information on 3D specific optimizations, see Optimizing 3D performance.

Reuse shaders and materials

The Godot renderer is a little different to what is out there. It's designed to minimize GPU state changes as much as possible. StandardMaterial3D does a good job at reusing materials that need similar shaders. If custom shaders are used, make sure to reuse them as much as possible. Godot's priorities are:

- **Reusing Materials:** The fewer different materials in the scene, the faster the rendering will be. If a scene has a huge amount of objects (in the hundreds or thousands), try reusing the materials. In the worst case, use atlases to decrease the amount of texture changes.
- **Reusing Shaders:** If materials can't be reused, at least try to reuse shaders. Note: shaders are automatically reused between StandardMaterial3Ds that share the same configuration

(features that are enabled or disabled with a check box) even if they have different parameters.

If a scene has, for example, 20,000 objects with 20,000 different materials each, rendering will be slow. If the same scene has 20,000 objects, but only uses 100 materials, rendering will be much faster.

Pixel cost versus vertex cost

You may have heard that the lower the number of polygons in a model, the faster it will be rendered. This is *really* relative and depends on many factors.

On a modern PC and console, vertex cost is low. GPUs originally only rendered triangles. This meant that every frame:

1. All vertices had to be transformed by the CPU (including clipping).
2. All vertices had to be sent to the GPU memory from the main RAM.

Nowadays, all this is handled inside the GPU, greatly increasing performance. 3D artists usually have the wrong feeling about polycount performance because 3D modeling software (such as Blender, 3ds Max, etc.) need to keep geometry in CPU memory for it to be edited, reducing actual performance. Game engines rely on the GPU more, so they can render many triangles much more efficiently.

On mobile devices, the story is different. PC and console GPUs are brute-force monsters that can pull as much electricity as they need from the power grid. Mobile GPUs are limited to a tiny battery, so they need to be a lot more power efficient.

To be more efficient, mobile GPUs attempt to avoid *overdraw*. Overdraw occurs when the same pixel on the screen is being rendered more than once. Imagine a town with several buildings.

GPUs don't know what is visible and what is hidden until they draw it. For example, a house might be drawn and then another house in front of it (which means rendering happened twice for the same pixel). PC GPUs normally don't care much about this and just throw more pixel processors to the hardware to increase performance (which also increases power consumption).

Using more power is not an option on mobile so mobile devices use a technique called *tile-based rendering* which divides the screen into a grid. Each cell keeps the list of triangles drawn to it and sorts them by depth to minimize *overdraw*. This technique improves performance and reduces power consumption, but takes a toll on vertex performance. As a result, fewer vertices and triangles can be processed for drawing.

Additionally, tile-based rendering struggles when there are small objects with a lot of geometry within a small portion of the screen. This forces mobile GPUs to put a lot of strain on a single screen tile, which considerably decreases performance as all the other cells must wait for it to complete before displaying the frame.

To summarize, don't worry about vertex count on mobile, but **avoid concentration of vertices in small parts of the screen**. If a character, NPC, vehicle, etc. is far away (which means it looks tiny), use a smaller level of detail (LOD) model. Even on desktop GPUs, it's preferable to avoid having triangles smaller than the size of a pixel on screen.

Pay attention to the additional vertex processing required when using:

- Skinning (skeletal animation)
- Morphs (shape keys)

Pixel/fragment shaders and fill rate

In contrast to vertex processing, the costs of fragment (per-pixel) shading have increased dramatically over the years. Screen resolutions have increased: the area of a 4K screen is 8,294,400 pixels, versus 307,200 for an old 640×480 VGA screen. That is 27 times the area! Also, the complexity of fragment shaders has exploded. Physically-based rendering requires complex calculations for each fragment.

You can test whether a project is fill rate-limited quite easily. Turn off V-Sync to prevent capping the frames per second, then compare the frames per second when running with a large window, to running with a very small window. You may also benefit from similarly reducing your shadow map size if using shadows. Usually, you will find the FPS increases quite a bit using a small window, which indicates you are to some extent fill rate-limited. On the other hand, if there is little to no increase in FPS, then your bottleneck lies elsewhere.

You can increase performance in a fill rate-limited project by reducing the amount of work the GPU has to do. You can do this by simplifying the shader (perhaps turn off expensive options if you are using a StandardMaterial3D), or reducing the number and size of textures used. Also, when using non-unshaded particles, consider forcing vertex shading in their material to decrease the shading cost.

See also

On supported hardware, Variable rate shading can be used to reduce shading processing costs without impacting the sharpness of edges on the final image.

When targeting mobile devices, consider using the simplest possible shaders you can reasonably afford to use.

Reading textures

The other factor in fragment shaders is the cost of reading textures. Reading textures is an expensive operation, especially when reading from several textures in a single fragment shader. Also, consider that filtering may slow it down further (trilinear filtering between mipmaps, and averaging). Reading textures is also expensive in terms of power usage, which is a big issue on mobiles.

If you use third-party shaders or write your own shaders, try to use algorithms that require as few texture reads as possible.

Texture compression

By default, Godot compresses textures of 3D models when imported using video RAM (VRAM) compression. Video RAM compression isn't as efficient in size as PNG or JPG when stored, but increases performance enormously when drawing large enough textures.

This is because the main goal of texture compression is bandwidth reduction between memory and the GPU.

In 3D, the shapes of objects depend more on the geometry than the texture, so compression is generally not noticeable. In 2D, compression depends more on shapes inside the textures, so the artifacts resulting from 2D compression are more noticeable.

As a warning, most Android devices do not support texture compression of textures with transparency (only opaque), so keep this in mind.

Note

Even in 3D, "pixel art" textures should have VRAM compression disabled as it will negatively affect their appearance, without improving performance significantly due to their low resolution.

Post-processing and shadows

Post-processing effects and shadows can also be expensive in terms of fragment shading activity. Always test the impact of these on different hardware.

Reducing the size of shadowmaps can increase performance, both in terms of writing and reading the shadowmaps. On top of that, the best way to improve performance of shadows is to turn shadows off for as many lights and objects as possible. Smaller or distant OmniLights/SpotLights can often have their shadows disabled with only a small visual impact.

Transparency and blending

Transparent objects present particular problems for rendering efficiency. Opaque objects (especially in 3D) can be essentially rendered in any order and the Z-buffer will ensure that only the front most objects get shaded. Transparent or blended objects are different. In most cases, they cannot rely on the Z-buffer and must be rendered in "painter's order" (i.e. from back to front) to look correct.

Transparent objects are also particularly bad for fill rate, because every item has to be drawn even if other transparent objects will be drawn on top later on.

Opaque objects don't have to do this. They can usually take advantage of the Z-buffer by writing to the Z-buffer only first, then only performing the fragment shader on the "winning" fragment, the object that is at the front at a particular pixel.

Transparency is particularly expensive where multiple transparent objects overlap. It is usually better to use transparent areas as small as possible to minimize these fill rate requirements, especially on mobile, where fill rate is very expensive. Indeed, in many situations, rendering more complex opaque geometry can end up being faster than using transparency to "cheat".

Multi-platform advice

If you are aiming to release on multiple platforms, test *early* and test *often* on all your platforms, especially mobile. Developing a game on desktop but attempting to port it to mobile at the last minute is a recipe for disaster.

In general, you should design your game for the lowest common denominator, then add optional enhancements for more powerful platforms. For example, you may want to use the Compatibility rendering method for both desktop and mobile platforms where you target both.

Mobile/tiled renderers

As described above, GPUs on mobile devices work in dramatically different ways from GPUs on desktop. Most mobile devices use tile renderers. Tile renderers split up the screen into regular-sized tiles that fit into super fast cache memory, which reduces the number of read/write operations to the main memory.

There are some downsides though. Tiled rendering can make certain techniques much more complicated and expensive to perform. Tiles that rely on the results of rendering in different tiles or on the results of earlier operations being preserved can be very slow. Be very careful to test the performance of shaders, viewport textures and post processing.

Optimization using MultiMeshes

For large amount of instances (in the thousands), that need to be constantly processed (and certain amount of control needs to be retained), using servers directly is the recommended optimization.

When the amount of objects reach the hundreds of thousands or millions, none of these approaches are efficient anymore. Still, depending on the requirements, there is one more optimization possible.

MultiMeshes

A MultiMesh is a single draw primitive that can draw up to millions of objects in one go. It's extremely efficient because it uses the GPU hardware to do this (in OpenGL ES 2.0, it's less efficient because there is no hardware support for it, though).

The only drawback is that there is no *screen* or *frustum* culling possible for individual instances. This means, that millions of objects will be *always* or *never* drawn, depending on the visibility of the whole MultiMesh. It is possible to provide a custom visibility rect for them, but it will always be *all-or-none* visibility.

If the objects are simple enough (just a couple of vertices), this is generally not much of a problem as most modern GPUs are optimized for this use case. A workaround is to create several MultiMeshes for different areas of the world.

It is also possible to execute some logic inside the vertex shader (using the `INSTANCE_ID` or `INSTANCE_CUSTOM` built-in constants). For an example of animating thousands of objects in a MultiMesh, see the Animating thousands of fish tutorial. Information to the shader can be provided via textures (there are floating-point Image formats which are ideal for this).

Another alternative is to use a GDExtension and C++, which should be extremely efficient (it's possible to set the entire state for all objects using linear memory via the RenderingServer.multimesh_set_buffer() function). This way, the array can be created with multiple threads, then set in one call, providing high cache efficiency.

Finally, it's not required to have all MultiMesh instances visible. The amount of visible ones can be controlled with the MultiMesh.visible_instance_count property. The typical workflow is to allocate the maximum amount of instances that will be used, then change the amount visible depending on how many are currently needed.

Multimesh example

Here is an example of using a MultiMesh from code. Languages other than GDScript may be more efficient for millions of objects, but for a few thousands, GDScript should be fine.

GDScript

```
extends MultiMeshInstance3D


func _ready():
    # Create the multimesh.
    multimesh = MultiMesh.new()
    # Set the format first.
    multimesh.transform_format = MultiMesh.TRANSFORM_3D
    # Then resize (otherwise, changing the format is not
allowed).
```

```
    multimesh.instance_count = 10000
    # Maybe not all of them should be visible at first.
    multimesh.visible_instance_count = 1000

    # Set the transform of the instances.
    for i in multimesh.visible_instance_count:
        multimesh.set_instance_transform(i,
Transform3D(Basis(), Vector3(i * 20, 0, 0)))
```

C#

```
using Godot;

public partial class MyMultiMeshInstance3D :
MultiMeshInstance3D
{
    public override void _Ready()
    {
        // Create the multimesh.
        Multimesh = new MultiMesh();
        // Set the format first.
        Multimesh.TransformFormat =
MultiMesh.TransformFormatEnum.Transform3D;
        // Then resize (otherwise, changing the format is not
allowed)
        Multimesh.InstanceCount = 1000;
        // Maybe not all of them should be visible at first.
        Multimesh.VisibleInstanceCount = 1000;

        // Set the transform of the instances.
        for (int i = 0; i < Multimesh.VisibleInstanceCount;
i++)
        {
            Multimesh.SetInstanceTransform(i, new
Transform3D(Basis.Identity, new Vector3(i * 20, 0, 0)));
        }
    }
}
```

Optimizing 3D performance

Culling

Godot will automatically perform view frustum culling in order to prevent rendering objects that are outside the viewport. This works well for games that take place in a small area, however things can quickly become problematic in larger levels.

Occlusion culling

Walking around a town for example, you may only be able to see a few buildings in the street you are in, as well as the sky and a few birds flying overhead. As far as a naive renderer is concerned however, you can still see the entire town. It won't just render the buildings in front of you, it will render the street behind that, with the people on that street, the buildings behind that. You quickly end up in situations where you are attempting to render 10× or 100× more than what is visible.

Things aren't quite as bad as they seem, because the Z-buffer usually allows the GPU to only fully shade the objects that are at the front. This is called *depth prepass* and is enabled by default in Godot when using the Forward+ or Compatibility rendering methods. However, unneeded objects are still reducing performance.

One way we can potentially reduce the amount to be rendered is to **take advantage of occlusion**. Godot 4.0 and later offers a new approach to occlusion culling using occluder nodes. See Occlusion culling for instructions on setting up occlusion culling in your scene.

Note

In some cases, you may have to adapt your level design to add more occlusion opportunities. For example, you may have to add more walls to prevent the player from seeing too far away, which would decrease performance due to the lost opportunities for occlusion culling.

Transparent objects

Godot sorts objects by Material and Shader to improve performance. This, however, can not be done with transparent objects. Transparent objects are rendered from back to front to make blending with what is behind work. As a result, **try to use as few transparent objects as possible**. If an object has a small section with transparency, try to make that section a separate surface with its own material.

For more information, see the GPU optimizations doc.

Level of detail (LOD)

In some situations, particularly at a distance, it can be a good idea to **replace complex geometry with simpler versions**. The end user will probably not be able to see much difference. Consider looking at a large number of trees in the far distance. There are several strategies for replacing models at varying distance. You could use lower poly models, or use transparency to simulate more complex geometry.

Godot 4 offers several ways to control level of detail:

- An automatic approach on mesh import using Mesh level of detail (LOD).

- A manual approach configured in the 3D node using Visibility ranges (HLOD).
- Decals and lights can also benefit from level of detail using their respective **Distance Fade** properties.

While they can be used independently, these approaches are most effective when used together. For example, you can set up visibility ranges to hide particle effects that are too far away from the player to notice. At the same time, you can rely on mesh LOD to make the particle effect's meshes rendered with less detail at a distance.

Visibility ranges are also a good way to set up *impostors* for distant geometry (see below).

Billboards and imposters

The simplest version of using transparency to deal with LOD is billboards. For example, you can use a single transparent quad to represent a tree at distance. This can be very cheap to render, unless of course, there are many trees in front of each other. In this case, transparency may start eating into fill rate (for more information on fill rate, see GPU optimization).

An alternative is to render not just one tree, but a number of trees together as a group. This can be especially effective if you can see an area but cannot physically approach it in a game.

You can make imposters by pre-rendering views of an object at different angles. Or you can even go one step further, and periodically re-render a view of an object onto a texture to be used as an imposter. At a distance, you need to move the viewer a considerable distance for the angle of view to change significantly. This can be complex to get working, but may be worth it depending on the type of project you are making.

Use instancing (MultiMesh)

If several identical objects have to be drawn in the same place or nearby, try using MultiMesh instead. MultiMesh allows the drawing of many thousands of objects at very little performance cost, making it ideal for flocks, grass, particles, and anything else where you have thousands of identical objects.

See also the Using MultiMesh documentation.

Bake lighting

Lighting objects is one of the most costly rendering operations. Realtime lighting, shadows (especially multiple lights), and global illumination are especially expensive. They may simply be too much for lower power mobile devices to handle.

Consider using baked lighting, especially for mobile. This can look fantastic, but has the downside that it will not be dynamic. Sometimes, this is a tradeoff worth making.

See Using Lightmap global illumination for instructions on using baked lightmaps. For best performance, you should set lights' bake mode to **Static** as opposed to the default **Dynamic**, as this will skip real-time lighting on meshes that have baked lighting.

The downside of lights with the **Static** bake mode is that they can't cast shadows onto meshes with baked lighting. This can make scenes with outdoor environments and dynamic objects look flat. A good balance between performance and quality is to keep **Dynamic** for the DirectionalLight3D node, and use **Static** for most (if not all) omni and spot lights.

Animation and skinning

Animation and vertex animation such as skinning and morphing can be very expensive on some platforms. You may need to lower the polycount considerably for animated models, or limit the number of them on screen at any given time. You can also reduce the animation rate for distant or occluded meshes, or pause the animation entirely if the player is unlikely to notice the animation being stopped.

The VisibleOnScreenEnabler3D and VisibleOnScreenNotifier3D nodes can be useful for this purpose.

Large worlds

If you are making large worlds, there are different considerations than what you may be familiar with from smaller games.

Large worlds may need to be built in tiles that can be loaded on demand as you move around the world. This can prevent memory use from getting out of hand, and also limit the processing needed to the local area.

There may also be rendering and physics glitches due to floating point error in large worlds. This can be resolved using Large world coordinates. If using large world coordinates is not an option, you may be able to use techniques such as orienting the world around the player (rather than the other way around), or shifting the origin periodically to keep things centred around `Vector3(0, 0, 0)`.

Animating thousands of objects

- Animating thousands of fish with MultiMeshInstance3D
- Controlling thousands of fish with Particles

Animating thousands of fish with MultiMeshInstance3D

This tutorial explores a technique used in the game ABZU [https://www.gdcvault.com/play/1024409/Creating-the-Art-of-ABZ] for rendering and animating thousands of fish using vertex animation and static mesh instancing.

In Godot, this can be accomplished with a custom Shader and a MultiMeshInstance3D. Using the following technique you can render thousands of animated objects, even on low end hardware.

We will start by animating one fish. Then, we will see how to extend that animation to thousands of fish.

Animating one Fish

We will start with a single fish. Load your fish model into a MeshInstance3D and add a new ShaderMaterial.

Here is the fish we will be using for the example images, you can use any fish model you like.

Note

The fish model in this tutorial is made by QuaterniusDev [https://quaternius.com] and is shared with a creative commons license. CC0 1.0 Universal (CC0 1.0) Public Domain Dedication https://creativecommons.org/publicdomain/zero/1.0/

Typically, you would use bones and a Skeleton3D to animate objects. However, bones are animated on the CPU and so you end having to calculate thousands of operations every frame and it becomes impossible to have thousands of objects. Using vertex animation in a vertex shader, you avoid using bones and can instead calculate the full animation in a few lines of code and completely on the GPU.

The animation will be made of four key motions:

1. A side to side motion
2. A pivot motion around the center of the fish
3. A panning wave motion
4. A panning twist motion

All the code for the animation will be in the vertex shader with uniforms controlling the amount of motion. We use uniforms to control the strength of the motion so that you can tweak the animation in editor and see the results in real time, without the shader having to recompile.

All the motions will be made using cosine waves applied to `VERTEX` in model space. We want the vertices to be in model space so that the motion is always relative to the orientation of the fish. For example, side-to-side will always move the fish back and forth in its left to right direction, instead of on the `x` axis in the world orientation.

In order to control the speed of the animation, we will start by defining our own time variable using `TIME`.

```
//time_scale is a uniform float
float time = TIME * time_scale;
```

The first motion we will implement is the side to side motion. It can be made by offsetting `VERTEX.x` by `cos` of `TIME`. Each time the mesh is rendered, all the vertices will move to the side by the amount of `cos(time)`.

```
//side_to_side is a uniform float
VERTEX.x += cos(time) * side_to_side;
```

The resulting animation should look something like this:

Next, we add the pivot. Because the fish is centered at (0, 0), all we have to do is multiply `VERTEX` by a rotation matrix for it to rotate around the center of the fish.

We construct a rotation matrix like so:

```
//angle is scaled by 0.1 so that the fish only pivots and
doesn't rotate all the way around
//pivot is a uniform float
float pivot_angle = cos(time) * 0.1 * pivot;
mat2 rotation_matrix = mat2(vec2(cos(pivot_angle), -
sin(pivot_angle)), vec2(sin(pivot_angle), cos(pivot_angle)));
```

And then we apply it in the `x` and `z` axes by multiplying it by `VERTEX.xz`.

```
VERTEX.xz = rotation_matrix * VERTEX.xz;
```

With only the pivot applied you should see something like this:

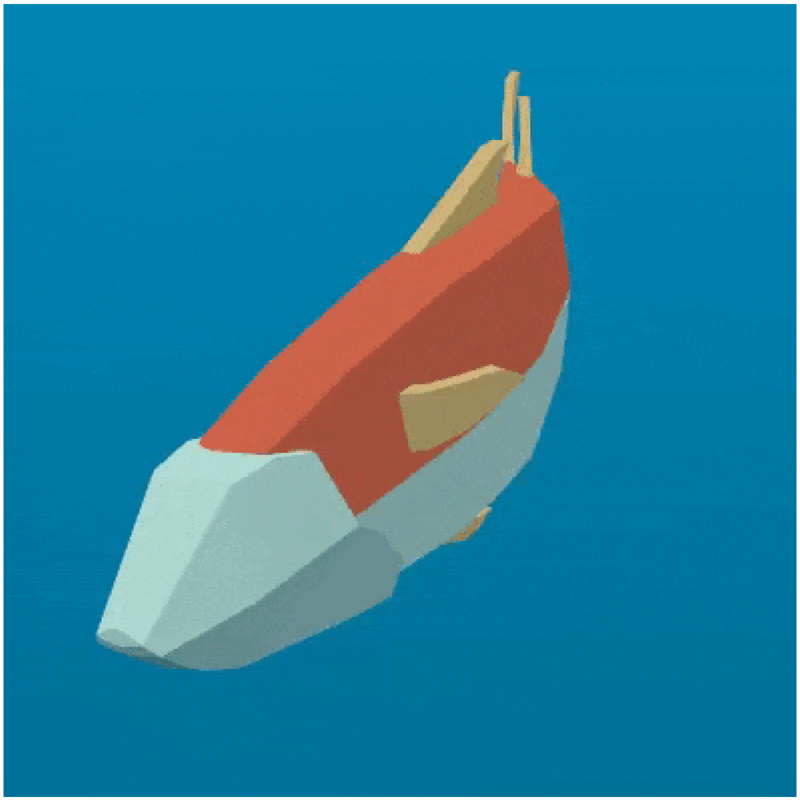
The next two motions need to pan down the spine of the fish. For that, we need a new variable, `body`. `body` is a float that is `0` at the tail of the fish and `1` at its head.

```
float body = (VERTEX.z + 1.0) / 2.0; //for a fish centered at
(0, 0) with a length of 2
```

The next motion is a cosine wave that moves down the length of the fish. To make it move along the spine of the fish, we offset the input to `cos` by the position along the spine, which is the variable we defined above, `body`.

```
//wave is a uniform float
VERTEX.x += cos(time + body) * wave;
```

This looks very similar to the side to side motion we defined above, but in this one, by using `body` to offset `cos` each vertex along the spine has a different position in the wave making it look like a wave is moving along the fish.



The last motion is the twist, which is a panning roll along the spine. Similarly to the pivot, we first construct a rotation matrix.

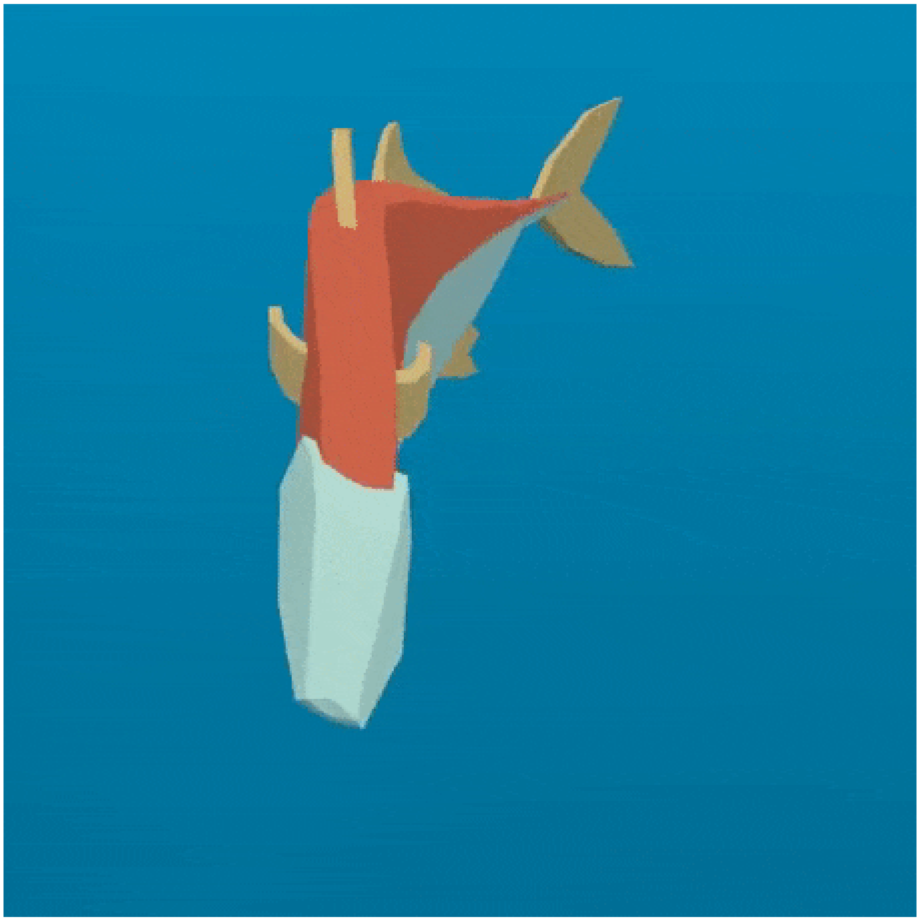
```
//twist is a uniform float
float twist_angle = cos(time + body) * 0.3 * twist;
mat2 twist_matrix = mat2(vec2(cos(twist_angle), -
sin(twist_angle)), vec2(sin(twist_angle), cos(twist_angle)));
```

We apply the rotation in the `xy` axes so that the fish appears to roll around its spine. For this to work, the fish's spine needs to be centered on the `z` axis.

```
VERTEX.xy = twist_matrix * VERTEX.xy;
```

Here is the fish with twist applied:

If we apply all these motions one after another, we get a fluid jelly-like motion.



Normal fish swim mostly with the back half of their body. Accordingly, we need to limit the panning motions to the back half of the fish. To do this, we create a new variable, `mask`.

`mask` is a float that goes from `0` at the front of the fish to `1` at the end using `smoothstep` to control the point at which the transition from `0` to `1` happens.

```
//mask_black and mask_white are uniforms
float mask = smoothstep(mask_black, mask_white, 1.0 - body);
```

Below is an image of the fish with `mask` used as `COLOR`:

For the wave, we multiply the motion by `mask` which will limit it to the back half.

```
//wave motion with mask
VERTEX.x += cos(time + body) * mask * wave;
```

In order to apply the mask to the twist, we use `mix`. `mix` allows us to mix the vertex position between a fully rotated vertex and one that is not rotated. We need to use `mix` instead of multiplying `mask` by the rotated `VERTEX` because we are not adding the motion to the `VERTEX` we are replacing the `VERTEX` with the rotated version. If we multiplied that by `mask`, we would shrink the fish.

```
//twist motion with mask
VERTEX.xy = mix(VERTEX.xy, twist_matrix * VERTEX.xy, mask);
```

Putting the four motions together gives us the final animation.

Go ahead and play with the uniforms in order to alter the swim cycle of the fish. You will find that you can create a wide variety of swim styles using these four motions.

Making a school of fish

Godot makes it easy to render thousands of the same object using a MultiMeshInstance3D node.

A MultiMeshInstance3D node is created and used the same way you would make a MeshInstance3D node. For this tutorial, we will name the MultiMeshInstance3D node `School`, because it will contain a school of fish.

Once you have a MultiMeshInstance3D add a MultiMesh, and to that MultiMesh add your Mesh with the shader from above.

MultiMeshes draw your Mesh with three additional per-instance properties: Transform (rotation, translation, scale), Color, and Custom. Custom is used to pass in 4 multi-use variables using a Color.

`instance_count` specifies how many instances of the mesh you want to draw. For now, leave `instance_count` at `0` because you cannot change any of the other parameters while `instance_count` is larger than `0`. We will set `instance count` in GDScript later.

`transform_format` specifies whether the transforms used are 3D or 2D. For this tutorial, select 3D.

For both `color_format` and `custom_data_format` you can choose between `None`, `Byte`, and `Float`. `None` means you won't be passing in that data (either a per-instance `COLOR` variable, or `INSTANCE_CUSTOM`) to the shader. `Byte` means each number making up the color you pass in will be stored with 8 bits while `Float` means each number will be stored in a floating-point number (32 bits). `Float` is slower but more precise, `Byte` will take less memory and be faster, but you may see some visual artifacts.

Now, set `instance_count` to the number of fish you want to have.

Next we need to set the per-instance transforms.

There are two ways to set per-instance transforms for MultiMeshes. The first is entirely in editor and is described in the MultiMeshInstance3D tutorial.

The second is to loop over all the instances and set their transforms in code. Below, we use GDScript to loop over all the instances and set their transform to a random position.

```
for i in range($School.multimesh.instance_count):
  var position = Transform3D()
  position = position.translated(Vector3(randf() * 100 - 50,
randf() * 50 - 25, randf() * 50 - 25))
  $School.multimesh.set_instance_transform(i, position)
```

Running this script will place the fish in random positions in a box around the position of the MultiMeshInstance3D.

Note

If performance is an issue for you, try running the scene with fewer fish.

Notice how all the fish are all in the same position in their swim cycle? It makes them look very robotic. The next step is to give each fish a different position in the swim cycle so the entire school looks more organic.

Animating a school of fish

One of the benefits of animating the fish using `cos` functions is that they are animated with one parameter, `time`. In order to give each fish a unique position in the swim cycle, we only need to offset `time`.

We do that by adding the per-instance custom value `INSTANCE_CUSTOM` to `time`.

```
float time = (TIME * time_scale) + (6.28318 *
INSTANCE_CUSTOM.x);
```

Next, we need to pass a value into `INSTANCE_CUSTOM`. We do that by adding one line into the `for` loop from above. In the `for` loop we assign each instance a set of four random floats to use.

```
$School.multimesh.set_instance_custom_data(i, Color(randf(),
randf(), randf(), randf()))
```

Now the fish all have unique positions in the swim cycle. You can give them a little more individuality by using `INSTANCE_CUSTOM` to make them swim faster or slower by multiplying by `TIME`.

```
//set speed from 50% - 150% of regular speed
float time = (TIME * (0.5 + INSTANCE_CUSTOM.y) * time_scale) +
(6.28318 * INSTANCE_CUSTOM.x);
```

You can even experiment with changing the per-instance color the same way you changed the per-instance custom value.

One problem that you will run into at this point is that the fish are animated, but they are not moving. You can move them by updating the per-instance transform for each fish every frame. Although doing so will be faster than moving thousands of MeshInstance3Ds per frame, it'll still likely be slow.

In the next tutorial we will cover how to use GPUParticles3D to take advantage of the GPU and move each fish around individually while still receiving the benefits of instancing.

Controlling thousands of fish with Particles

The problem with MeshInstance3D is that it is expensive to update their transform array. It is great for placing many static objects around the scene. But it is still difficult to move the objects around the scene.

To make each instance move in an interesting way, we will use a GPUParticles3D node. Particles take advantage of GPU acceleration by computing and setting the per-instance information in a Shader.

First create a Particles node. Then, under "Draw Passes" set the Particle's "Draw Pass 1" to your Mesh. Then under "Process Material" create a new ShaderMaterial.

Set the `shader_type` to `particles`.

```
shader_type particles
```

Then add the following two functions:

```
float rand_from_seed(in uint seed) {
  int k;
  int s = int(seed);
  if (s == 0)
    s = 305420679;
  k = s / 127773;
  s = 16807 * (s - k * 127773) - 2836 * k;
  if (s < 0)
    s += 2147483647;
  seed = uint(s);
  return float(seed % uint(65536)) / 65535.0;
}

uint hash(uint x) {
  x = ((x >> uint(16)) ^ x) * uint(73244475);
  x = ((x >> uint(16)) ^ x) * uint(73244475);
  x = (x >> uint(16)) ^ x;
  return x;
}
```

These functions come from the default ParticleProcessMaterial. They are used to generate a random number from each particle's `RANDOM_SEED`.

A unique thing about particle shaders is that some built-in variables are saved across frames. `TRANSFORM`, `COLOR`, and `CUSTOM` can all be accessed in the shader of the mesh, and also in the particle shader the next time it is run.

Next, setup your `start()` function. Particles shaders contain a `start()` function and a `process()` function.

The code in the `start()` function only runs when the particle system starts. The code in the `process()` function will always run.

We need to generate 4 random numbers: 3 to create a random position and one for the random offset of the swim cycle.

First, generate 4 seeds inside the `start()` function using the `hash()` function provided above:

```
uint alt_seed1 = hash(NUMBER + uint(1) + RANDOM_SEED);
uint alt_seed2 = hash(NUMBER + uint(27) + RANDOM_SEED);
uint alt_seed3 = hash(NUMBER + uint(43) + RANDOM_SEED);
uint alt_seed4 = hash(NUMBER + uint(111) + RANDOM_SEED);
```

Then, use those seeds to generate random numbers using `rand_from_seed`:

```
CUSTOM.x = rand_from_seed(alt_seed1);
vec3 position = vec3(rand_from_seed(alt_seed2) * 2.0 - 1.0,
                     rand_from_seed(alt_seed3) * 2.0 - 1.0,
                     rand_from_seed(alt_seed4) * 2.0 - 1.0);
```

Finally, assign `position` to `TRANSFORM[3].xyz`, which is the part of the transform that holds the position information.

```
TRANSFORM[3].xyz = position * 20.0;
```

Remember, all this code so far goes inside the `start()` function.

The vertex shader for your mesh can stay the exact same as it was in the previous tutorial.

Now you can move each fish individually each frame, either by adding to the `TRANSFORM` directly or by writing to `VELOCITY`.

Let's transform the fish by setting their `VELOCITY` in the `start()` function.

```
VELOCITY.z = 10.0;
```

This is the most basic way to set `VELOCITY` every particle (or fish) will have the same velocity.

Just by setting `VELOCITY` you can make the fish swim however you want. For example, try the code below.

```
VELOCITY.z = cos(TIME + CUSTOM.x * 6.28) * 4.0 + 6.0;
```

This will give each fish a unique speed between `2` and `10`.

You can also let each fish change its velocity over time if you set the velocity in the `process()` function.

If you used `CUSTOM.y` in the last tutorial, you can also set the speed of the swim animation based on the `VELOCITY`. Just use `CUSTOM.y`.

```
CUSTOM.y = VELOCITY.z * 0.1;
```

This code gives you the following behavior:

Using a ParticleProcessMaterial you can make the fish behavior as simple or complex as you like. In this tutorial we only set Velocity, but in your own Shaders you can also set `COLOR`, rotation, scale (through `TRANSFORM`). Please refer to the Particles Shader Reference for more information on particle shaders.

Using multiple threads

Threads

Threads allow simultaneous execution of code. It allows off-loading work from the main thread.

Godot supports threads and provides many handy functions to use them.

Note

If using other languages (C#, C++), it may be easier to use the threading classes they support.

Warning

Before using a built-in class in a thread, read Thread-safe APIs first to check whether it can be safely used in a thread.

Creating a Thread

To create a thread, use the following code:

GDScript

```
var thread: Thread

# The thread will start here.
func _ready():
    thread = Thread.new()
    # You can bind multiple arguments to a function Callable.
    thread.start(_thread_function.bind("Wafflecopter"))


# Run here and exit.
# The argument is the bound data passed from start().
func _thread_function(userdata):
    # Print the userdata ("Wafflecopter")
    print("I'm a thread! Userdata is: ", userdata)


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    thread.wait_to_finish()
```

Your function will, then, run in a separate thread until it returns. Even if the function has returned already, the thread must collect it, so call Thread.wait_to_finish(), which will wait until the thread is done (if not done yet), then properly dispose of it.

Warning

Creating threads at run-time is slow on Windows and should be avoided to prevent stuttering. Semaphores, explained later on this page, should be used instead.

Mutexes

Accessing objects or data from multiple threads is not always supported (if you do it, it will cause unexpected behaviors or crashes). Read the Thread-safe APIs documentation to understand which engine APIs support multiple thread access.

When processing your own data or calling your own functions, as a rule, try to avoid accessing the same data directly from different threads. You may run into synchronization problems, as the data is not always updated between CPU cores when modified. Always use a Mutex when accessing a piece of data from different threads.

When calling Mutex.lock(), a thread ensures that all other threads will be blocked (put on suspended state) if they try to *lock* the same mutex. When the mutex is unlocked by calling Mutex.unlock(), the other threads will be allowed to proceed with the lock (but only one at a time).

Here is an example of using a Mutex:

GDScript

```gdscript
var counter := 0
var mutex: Mutex
var thread: Thread


# The thread will start here.
func _ready():
    mutex = Mutex.new()
    thread = Thread.new()
    thread.start(_thread_function)

    # Increase value, protect it with Mutex.
    mutex.lock()
    counter += 1
    mutex.unlock()


# Increment the value from the thread, too.
func _thread_function():
    mutex.lock()
    counter += 1
    mutex.unlock()


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    thread.wait_to_finish()
    print("Counter is: ", counter) # Should be 2.
```

Semaphores

Sometimes you want your thread to work *"on demand"*. In other words, tell it when to work and let it suspend when it isn't doing anything. For this, Semaphores are used. The function Semaphore.wait() is used in the thread to suspend it until some data arrives.

The main thread, instead, uses Semaphore.post() to signal that data is ready to be processed:

GDScript

```
var counter := 0
var mutex: Mutex
var semaphore: Semaphore
var thread: Thread
var exit_thread := false


# The thread will start here.
func _ready():
    mutex = Mutex.new()
    semaphore = Semaphore.new()
    exit_thread = false

    thread = Thread.new()
    thread.start(_thread_function)


func _thread_function():
    while true:
        semaphore.wait() # Wait until posted.

        mutex.lock()
        var should_exit = exit_thread # Protect with Mutex.
        mutex.unlock()

        if should_exit:
            break

        mutex.lock()
        counter += 1 # Increment counter, protect with Mutex.
        mutex.unlock()


func increment_counter():
    semaphore.post() # Make the thread process.


func get_counter():
    mutex.lock()
    # Copy counter, protect with Mutex.
    var counter_value = counter
    mutex.unlock()
    return counter_value


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    # Set exit condition to true.
    mutex.lock()
    exit_thread = true # Protect with Mutex.
    mutex.unlock()

    # Unblock by posting.
    semaphore.post()

    # Wait until it exits.
    thread.wait_to_finish()

    # Print the counter.
    print("Counter is: ", counter)
```

Thread-safe APIs

Threads

Threads are used to balance processing power across CPUs and cores. Godot supports multithreading, but not in the whole engine.

Below is a list of ways multithreading can be used in different areas of Godot.

Global scope

Global Scope singletons are all thread-safe. Accessing servers from threads is supported (for RenderingServer and Physics servers, ensure threaded or thread-safe operation is enabled in the project settings!).

This makes them ideal for code that creates dozens of thousands of instances in servers and controls them from threads. Of course, it requires a bit more code, as this is used directly and not within the scene tree.

Scene tree

Interacting with the active scene tree is **NOT** thread-safe. Make sure to use mutexes when sending data between threads. If you want to call functions from a thread, the *call_deferred* function may be used:

```
# Unsafe:
node.add_child(child_node)
```

```
# Safe:
node.add_child.call_deferred(child_node)
```

However, creating scene chunks (nodes in tree arrangement) outside the active tree is fine. This way, parts of a scene can be built or instantiated in a thread, then added in the main thread:

```
var enemy_scene = load("res://enemy_scene.scn")
var enemy = enemy_scene.instantiate()
enemy.add_child(weapon) # Set a weapon.
world.add_child.call_deferred(enemy)
```

Still, this is only really useful if you have **one** thread loading data. Attempting to load or create scene chunks from multiple threads may work, but you risk resources (which are only loaded once in Godot) tweaked by the multiple threads, resulting in unexpected behaviors or crashes.

Only use more than one thread to generate scene data if you *really* know what you are doing and you are sure that a single resource is not being used or set in multiple ones. Otherwise, you are safer just using the servers API (which is fully thread-safe) directly and not touching scene or resources.

Rendering

Instancing nodes that render anything in 2D or 3D (such as Sprite) is *not* thread-safe by default. To make rendering thread-safe, set the **Rendering > Driver > Thread Model** project setting to **Multi-Threaded**.

Note that the Multi-Threaded thread model has several known bugs, so it may not be usable in all scenarios.

You should avoid calling functions involving direct interaction with the GPU on other threads, such as creating new textures or modifying and retrieving image data, these operations can lead to performance stalls because they require synchronization with the RenderingServer, as data needs to be transmitted to or updated on the GPU.

GDScript arrays, dictionaries

In GDScript, reading and writing elements from multiple threads is OK, but anything that changes the container size (resizing, adding or removing elements) requires locking a mutex.

Resources

Modifying a unique resource from multiple threads is not supported. However handling references on multiple threads is supported, hence loading resources on a thread is as well - scenes, textures, meshes, etc - can be loaded and manipulated on a thread and then added to the active scene on the main thread. The limitation here is as described above, one must be careful not to load the same resource from multiple threads at once, therefore it is easiest to use **one** thread for loading and modifying resources, and then the main thread for adding them.

Physics

- Physics introduction
- Using RigidBody
- Using Area2D
- Using CharacterBody2D/3D
- Ray-casting
- Ragdoll system
- Kinematic character (2D)
- Using SoftBody
- Collision shapes (2D)
- Collision shapes (3D)
- Large world coordinates
- Troubleshooting physics issues

Physics introduction

In game development, you often need to know when two objects in the game intersect or come into contact. This is known as **collision detection**. When a collision is detected, you typically want something to happen. This is known as **collision response**.

Godot offers a number of collision objects in 2D and 3D to provide both collision detection and response. Trying to decide which one to use for your project can be confusing. You can avoid problems and simplify development if you understand how each works and what their pros and cons are.

In this guide, you will learn:

- Godot's four collision object types
- How each collision object works
- When and why to choose one type over another

Note

This document's examples will use 2D objects. Every 2D physics object and collision shape has a direct equivalent in 3D and in most cases they work in much the same way.

Collision objects

Godot offers four kinds of collision objects which all extend CollisionObject2D. The last three listed below are physics bodies and additionally extend PhysicsBody2D.

- Area2D

  `Area2D` nodes provide **detection** and **influence**. They can detect when objects overlap and can emit signals when bodies enter or exit. An `Area2D` can also be used to override physics properties, such as gravity or damping, in a defined area.

- StaticBody2D

  A static body is one that is not moved by the physics engine. It participates in collision detection, but does not move in response to the collision. They are most often used for objects that are part of the environment or that do not need to have any dynamic behavior.

- RigidBody2D

  This is the node that implements simulated 2D physics. You do not control a `RigidBody2D` directly, but instead you apply forces to it (gravity, impulses, etc.) and the physics engine calculates the resulting movement. Read more about using rigid bodies.

- CharacterBody2D

  A body that provides collision detection, but no physics. All movement and collision response must be implemented in code.

Physics material

Static bodies and rigid bodies can be configured to use a PhysicsMaterial. This allows adjusting the friction and bounce of an object, and set if it's absorbent and/or rough.

Collision shapes

A physics body can hold any number of Shape2D objects as children. These shapes are used to define the object's collision bounds and to detect contact with other objects.

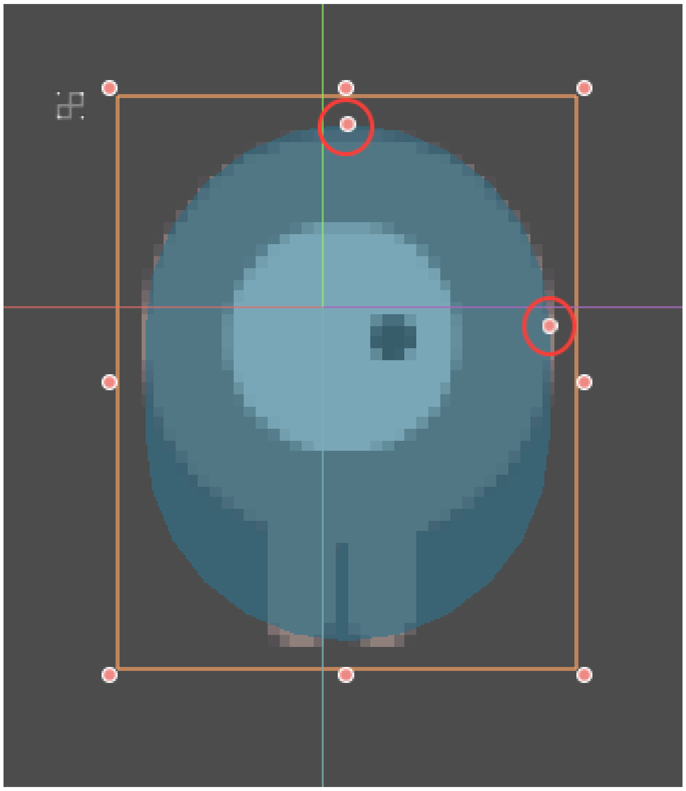
Note

In order to detect collisions, at least one `Shape2D` must be assigned to the object.

The most common way to assign a shape is by adding a CollisionShape2D or CollisionPolygon2D as a child of the object. These nodes allow you to draw the shape directly in the editor workspace.

Important

Be careful to never scale your collision shapes in the editor. The "Scale" property in the Inspector should remain `(1, 1)`. When changing the size of the collision shape, you should always use the size handles, **not** the `Node2D` scale handles. Scaling a shape can result in unexpected collision behavior.



Physics process callback

The physics engine runs at a fixed rate (a default of 60 iterations per second). This rate is typically different from the frame rate which fluctuates based on what is rendered and available resources.

It is important that all physics related code runs at this fixed rate. Therefore Godot differentiates [between physics and idle processing](). Code that runs each frame is called idle processing and code that runs on each physics tick is called physics processing. Godot provides two different callbacks, one for each of those processing rates.

The physics callback, [Node._physics_process()](), is called before each physics step. Any code that needs to access a body's properties should be run in here. This method will be passed a `delta` parameter, which is a floating-point number equal to the time passed in *seconds* since the last step. When using the default 60 Hz physics update rate, it will typically be equal to `0.01666...` (but not always, see below).

Note

It's recommended to always use the `delta` parameter when relevant in your physics calculations, so that the game behaves correctly if you change the physics update rate or if the player's device can't keep up.

Collision layers and masks

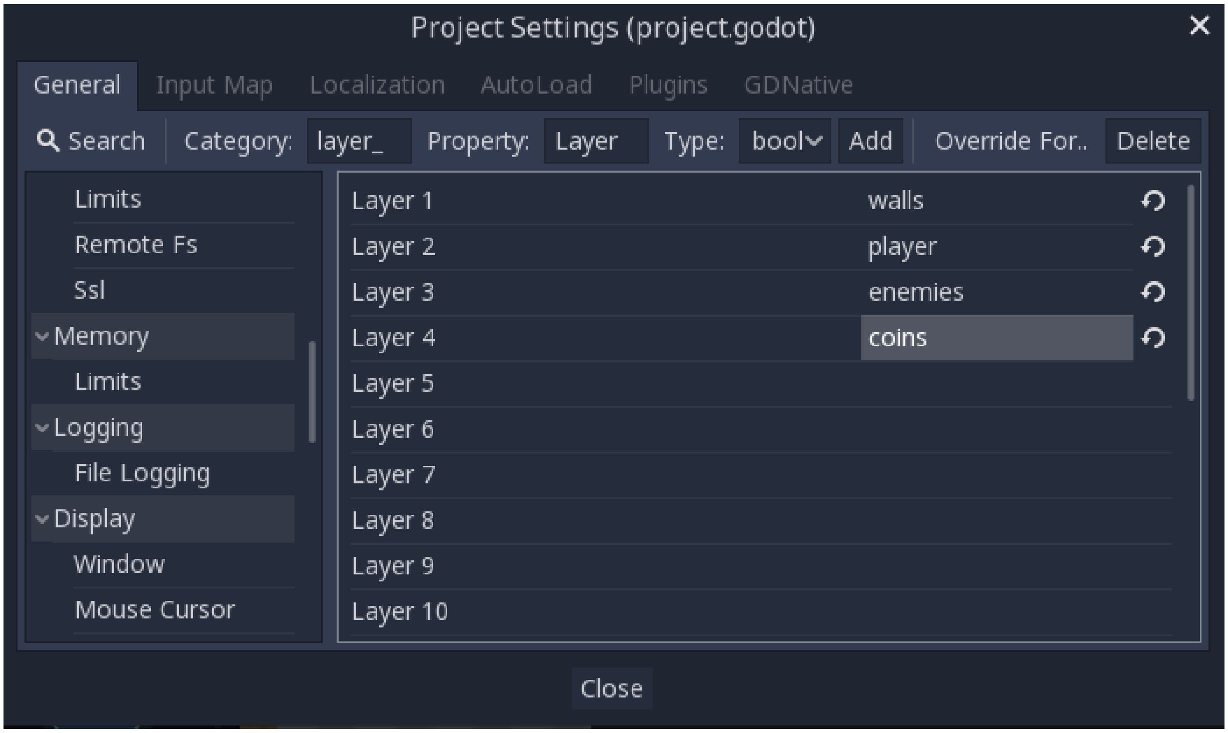
One of the most powerful, but frequently misunderstood, collision features is the collision layer system. This system allows you to build up complex interactions between a variety of objects. The key concepts are **layers** and **masks**. Each `CollisionObject2D` has 32 different physics layers it can interact with.

Let's look at each of the properties in turn:

- collision_layer
  This describes the layers that the object appears **in**. By default, all bodies are on layer `1`.
- collision_mask
  This describes what layers the body will **scan** for collisions. If an object isn't in one of the mask layers, the body will ignore it. By default, all bodies scan layer `1`.

These properties can be configured via code, or by editing them in the Inspector.

Keeping track of what you're using each layer for can be difficult, so you may find it useful to assign names to the layers you're using. Names can be assigned in Project Settings -> Layer Names.

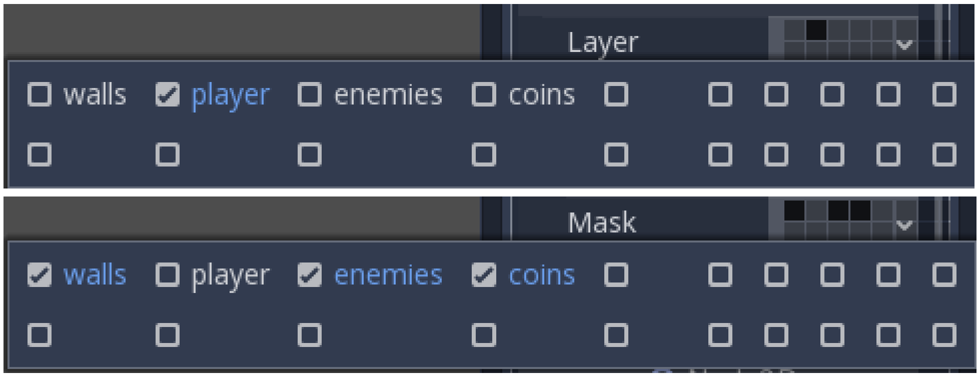

Project Settings (project.godot)
General
Input Map
Localization
AutoLoad
Plugins
GDNative
Search
Category:
layer_
Property:
Layer
Type:
bool
Add
Override For..
Delete
Limits
Remote Fs
Ssl
Memory
Limits
Logging
File Logging
Display
Window
Mouse Cursor
Close

Layer 1	walls
Layer 2	player
Layer 3	enemies
Layer 4	coins
Layer 5	
Layer 6	
Layer 7	
Layer 8	
Layer 9	
Layer 10	

GUI example

You have four node types in your game: Walls, Player, Enemy, and Coin. Both Player and Enemy should collide with Walls. The Player node should detect collisions with both Enemy and Coin, but Enemy and Coin should ignore each other.

Start by naming layers 1-4 "walls", "player", "enemies", and "coins" and place each node type in its respective layer using the "Layer" property. Then set each node's "Mask" property by selecting the layers it should interact with. For example, the Player's settings would look like this:


Layer

☐ walls ☑ player ☐ enemies ☐ coins ☐ ☐ ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

Mask

☑ walls ☐ player ☑ enemies ☑ coins ☐ ☐ ☐ ☐ ☐ ☐
☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

Code example

In function calls, layers are specified as a bitmask. Where a function enables all layers by default, the layer mask will be given as `0xffffffff`. Your code can use binary, hexadecimal, or decimal notation for layer masks, depending on your preference.

The code equivalent of the above example where layers 1, 3 and 4 were enabled would be as follows:

```
# Example: Setting mask value for enabling layers 1, 3 and 4

# Binary - set the bit corresponding to the layers you want to
enable (1, 3, and 4) to 1, set all other bits to 0.
# Note: Layer 32 is the first bit, layer 1 is the last. The
mask for layers 4,3 and 1 is therefore
0b00000000_00000000_00000000_00001101
# (This can be shortened to 0b1101)

# Hexadecimal equivalent (1101 binary converted to hexadecimal)
0x000d
# (This value can be shortened to 0xd)

# Decimal - Add the results of 2 to the power of (layer to be
enabled - 1).
# (2^(1-1)) + (2^(3-1)) + (2^(4-1)) = 1 + 4 + 8 = 13
pow(2, 1-1) + pow(2, 3-1) + pow(2, 4-1)
```

Area2D

Area nodes provide **detection** and **influence**. They can detect when objects overlap and emit signals when bodies enter or exit. Areas can also be used to override physics properties, such as gravity or damping, in a defined area.

There are three main uses for Area2D:

- Overriding physics parameters (such as gravity) in a given region.
- Detecting when other bodies enter or exit a region or what bodies are currently in a region.
- Checking other areas for overlap.

By default, areas also receive mouse and touchscreen input.

StaticBody2D

A static body is one that is not moved by the physics engine. It participates in collision detection, but does not move in response to the collision. However, it can impart motion or rotation to a colliding body **as if** it were moving, using its `constant_linear_velocity` and `constant_angular_velocity` properties.

`StaticBody2D` nodes are most often used for objects that are part of the environment or that do not need to have any dynamic behavior.

Example uses for `StaticBody2D`:

- Platforms (including moving platforms)
- Conveyor belts
- Walls and other obstacles

RigidBody2D

This is the node that implements simulated 2D physics. You do not control a RigidBody2D directly. Instead, you apply forces to it and the physics engine calculates the resulting movement, including collisions with other bodies, and collision responses, such as bouncing, rotating, etc.

You can modify a rigid body's behavior via properties such as "Mass", "Friction", or "Bounce", which can be set in the Inspector.

The body's behavior is also affected by the world's properties, as set in *Project Settings -> Physics*, or by entering an Area2D that is overriding the global physics properties.

When a rigid body is at rest and hasn't moved for a while, it goes to sleep. A sleeping body acts like a static body, and its forces are not calculated by the physics engine. The body will wake up when forces are applied, either by a collision or via code.

Using RigidBody2D

One of the benefits of using a rigid body is that a lot of behavior can be had "for free" without writing any code. For example, if you were making an "Angry Birds"-style game with falling blocks, you would only need to create RigidBody2Ds and adjust their properties. Stacking, falling, and bouncing would automatically be calculated by the physics engine.

However, if you do wish to have some control over the body, you should take care - altering the `position`, `linear_velocity`, or other physics properties of a rigid body can result in unexpected behavior. If you need to alter any of the physics-related properties, you should use the _integrate_forces() callback instead of `_physics_process()`. In this callback, you have access to the body's PhysicsDirectBodyState2D, which allows for safely changing properties and synchronizing them with the physics engine.

For example, here is the code for an "Asteroids" style spaceship:

GDScript

```
extends RigidBody2D

var thrust = Vector2(0, -250)
var torque = 20000

func _integrate_forces(state):
    if Input.is_action_pressed("ui_up"):
        state.apply_force(thrust.rotated(rotation))
    else:
        state.apply_force(Vector2())
    var rotation_direction = 0
    if Input.is_action_pressed("ui_right"):
        rotation_direction += 1
    if Input.is_action_pressed("ui_left"):
        rotation_direction -= 1
    state.apply_torque(rotation_direction * torque)
```

C#

```
using Godot;

public partial class Spaceship : RigidBody2D
{
    private Vector2 _thrust = new Vector2(0, -250);
    private float _torque = 20000;

    public override void
_IntegrateForces(PhysicsDirectBodyState2D state)
    {
        if (Input.IsActionPressed("ui_up"))
            state.ApplyForce(_thrust.Rotated(Rotation));
        else
            state.ApplyForce(new Vector2());

        var rotationDir = 0;
        if (Input.IsActionPressed("ui_right"))
            rotationDir += 1;
        if (Input.IsActionPressed("ui_left"))
```

```
            rotationDir -= 1;
        state.ApplyTorque(rotationDir * _torque);
    }
}
```

Note that we are not setting the `linear_velocity` or `angular_velocity` properties directly, but rather applying forces (`thrust` and `torque`) to the body and letting the physics engine calculate the resulting movement.

Note

When a rigid body goes to sleep, the `_integrate_forces()` function will not be called. To override this behavior, you will need to keep the body awake by creating a collision, applying a force to it, or by disabling the can_sleep property. Be aware that this can have a negative effect on performance.

Contact reporting

By default, rigid bodies do not keep track of contacts, because this can require a huge amount of memory if many bodies are in the scene. To enable contact reporting, set the max_contacts_reported property to a non-zero value. The contacts can then be obtained via PhysicsDirectBodyState2D.get_contact_count() and related functions.

Contact monitoring via signals can be enabled via the contact_monitor property. See RigidBody2D for the list of available signals.

CharacterBody2D

CharacterBody2D bodies detect collisions with other bodies, but are not affected by physics properties like gravity or friction. Instead, they must be controlled by the user via code. The physics engine will not move a character body.

When moving a character body, you should not set its `position` directly. Instead, you use the `move_and_collide()` or `move_and_slide()` methods. These methods move the body along a given vector, and it will instantly stop if a collision is detected with another body. After the body has collided, any collision response must be coded manually.

Character collision response

After a collision, you may want the body to bounce, to slide along a wall, or to alter the properties of the object it hit. The way you handle collision response depends on which method you used to move the CharacterBody2D.

move_and_collide

When using `move_and_collide()`, the function returns a KinematicCollision2D object, which contains information about the collision and the colliding body. You can use this information to determine the response.

For example, if you want to find the point in space where the collision occurred:

GDScript

```
extends PhysicsBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        var collision_point = collision_info.get_position()
```

C#

```
using Godot;

public partial class Body : PhysicsBody2D
{
    private Vector2 _velocity = new Vector2(250, 250);

    public override void _PhysicsProcess(double delta)
    {
        var collisionInfo = MoveAndCollide(_velocity *
(float)delta);
        if (collisionInfo != null)
        {
            var collisionPoint = collisionInfo.GetPosition();
        }
    }
}
```

Or to bounce off of the colliding object:

GDScript

```
extends PhysicsBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        velocity = velocity.bounce(collision_info.get_normal())
```

C#

```
using Godot;

public partial class Body : PhysicsBody2D
```

```
{
    private Vector2 _velocity = new Vector2(250, 250);

    public override void _PhysicsProcess(double delta)
    {
        var collisionInfo = MoveAndCollide(_velocity *
(float)delta);
        if (collisionInfo != null)
            _velocity =
_velocity.Bounce(collisionInfo.GetNormal());
    }
}
```

move_and_slide

Sliding is a common collision response; imagine a player moving along walls in a top-down game or running up and down slopes in a platformer. While it's possible to code this response yourself after using `move_and_collide()`, `move_and_slide()` provides a convenient way to implement sliding movement without writing much code.

Warning

`move_and_slide()` automatically includes the timestep in its calculation, so you should **not** multiply the velocity vector by `delta`. This does **not** apply to `gravity` as it is an acceleration and is time dependent, and needs to be scaled by `delta`.

For example, use the following code to make a character that can walk along the ground (including slopes) and jump when standing on the ground:

GDScript

```
extends CharacterBody2D
```

```
var run_speed = 350
var jump_speed = -1000
var gravity = 2500

func get_input():
    velocity.x = 0
    var right = Input.is_action_pressed('ui_right')
    var left = Input.is_action_pressed('ui_left')
    var jump = Input.is_action_just_pressed('ui_select')

    if is_on_floor() and jump:
        velocity.y = jump_speed
    if right:
        velocity.x += run_speed
    if left:
        velocity.x -= run_speed

func _physics_process(delta):
    velocity.y += gravity * delta
    get_input()
    move_and_slide()
```

C#

```
using Godot;

public partial class Body : CharacterBody2D
{
    private float _runSpeed = 350;
    private float _jumpSpeed = -1000;
    private float _gravity = 2500;

    private void GetInput()
    {
        var velocity = Velocity;
        velocity.X = 0;

        var right = Input.IsActionPressed("ui_right");
        var left = Input.IsActionPressed("ui_left");
        var jump = Input.IsActionPressed("ui_select");

        if (IsOnFloor() && jump)
```

```
            velocity.Y = _jumpSpeed;
        if (right)
            velocity.X += _runSpeed;
        if (left)
            velocity.X -= _runSpeed;

        Velocity = velocity;
    }

    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;
        velocity.Y += _gravity * (float)delta;
        Velocity = velocity;
        GetInput();
        MoveAndSlide();
    }
}
```

See Kinematic character (2D) for more details on using `move_and_slide()`, including a demo project with detailed code.

Using RigidBody

What is a rigid body?

A rigid body is one that is directly controlled by the physics engine in order to simulate the behavior of physical objects. In order to define the shape of the body, it must have one or more Shape3D objects assigned. Note that setting the position of these shapes will affect the body's center of mass.

How to control a rigid body

A rigid body's behavior can be altered by setting its properties, such as mass and weight. A physics material needs to be added to the rigid body to adjust its friction and bounce, and set if it's absorbent and/or rough. These properties can be set in the Inspector or via code. See RigidBody3D and PhysicsMaterial for the full list of properties and their effects.

There are several ways to control a rigid body's movement, depending on your desired application.

If you only need to place a rigid body once, for example to set its initial location, you can use the methods provided by the Node3D node, such as `set_global_transform()` or `look_at()`. However, these methods cannot be called every frame or the physics engine will not be able to correctly simulate the body's state. As an example, consider a rigid body that you want to rotate so that it points towards another object. A common mistake when implementing this kind of behavior is to use `look_at()` every frame, which breaks the physics simulation. Below, we'll demonstrate how to implement this correctly.

The fact that you can't use `set_global_transform()` or `look_at()` methods doesn't mean that you can't have full control of a rigid body. Instead, you can control it by using the `_integrate_forces()` callback. In this method, you can add *forces*, apply *impulses*, or set the *velocity* in order to achieve any movement you desire.

The "look at" method

As described above, using the Node3D's `look_at()` method can't be used each frame to follow a target. Here is a custom `look_at()` method called `look_follow()` that will work with rigid bodies:

GDScript

```
extends RigidBody3D

var speed: float = 0.1

func look_follow(state: PhysicsDirectBodyState3D,
current_transform: Transform3D, target_position: Vector3) ->
void:
    var forward_local_axis: Vector3 = Vector3(1, 0, 0)
    var forward_dir: Vector3 = (current_transform.basis *
forward_local_axis).normalized()
    var target_dir: Vector3 = (target_position -
current_transform.origin).normalized()
    var local_speed: float = clampf(speed, 0,
acos(forward_dir.dot(target_dir)))
    if forward_dir.dot(target_dir) > 1e-4:
        state.angular_velocity = local_speed *
forward_dir.cross(target_dir) / state.step

func _integrate_forces(state):
    var target_position =
$my_target_node3d_node.global_transform.origin
    look_follow(state, global_transform, target_position)
```

C#

```
using Godot;

public partial class MyRigidBody3D : RigidBody3D
{
    private float _speed = 0.1f;
    private void LookFollow(PhysicsDirectBodyState3D state,
Transform3D currentTransform, Vector3 targetPosition)
    {
        Vector3 forwardLocalAxis = new Vector3(1, 0, 0);
        Vector3 forwardDir = (currentTransform.Basis *
forwardLocalAxis).Normalized();
        Vector3 targetDir = (targetPosition -
currentTransform.Origin).Normalized();
        float localSpeed = Mathf.Clamp(_speed, 0.0f,
Mathf.Acos(forwardDir.Dot(targetDir)));
        if (forwardDir.Dot(targetDir) > 1e-4)
        {
            state.AngularVelocity = forwardDir.Cross(targetDir)
* localSpeed / state.Step;
        }
    }

    public override void
_IntegrateForces(PhysicsDirectBodyState3D state)
    {
        Vector3 targetPosition = GetNode<Node3D>
("MyTargetNode3DNode").GlobalTransform.Origin;
        LookFollow(state, GlobalTransform, targetPosition);
    }
}
```

This method uses the rigid body's `angular_velocity` property to rotate the body. The axis to rotate around is given by the cross product between the current forward direction and the direction one wants to look in. The `clamp` is a simple method used to prevent the amount of rotation from going past the direction which is wanted to be looked in, as the total amount of rotation needed is given by the arccosine of the dot product. This method can be used with `axis_lock_angular_*` as well. If more precise control is needed, solutions such as ones relying on Quaternion may be required, as discussed in Using 3D transforms.

Using Area2D

Introduction

Godot offers a number of collision objects to provide both collision detection and response. Trying to decide which one to use for your project can be confusing. You can avoid problems and simplify development if you understand how each of them works and what their pros and cons are. In this tutorial, we'll look at the Area2D node and show some examples of how it can be used.

Note

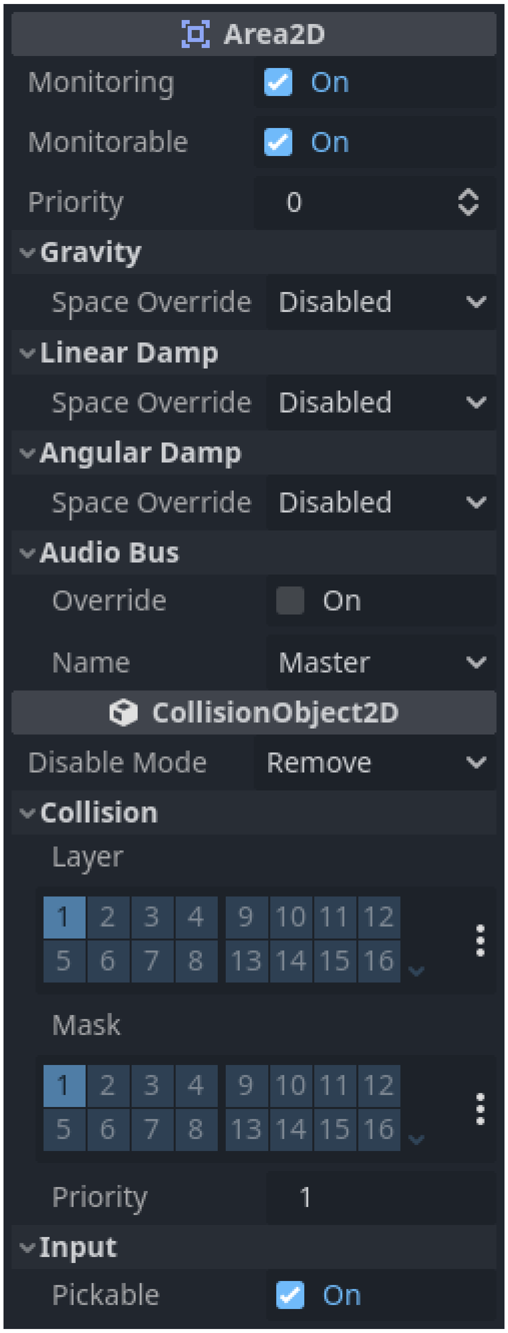
This document assumes you're familiar with Godot's various physics bodies. Please read Physics introduction first.

What is an area?

An Area2D defines a region of 2D space. In this space you can detect other CollisionObject2D nodes overlapping, entering, and exiting. Areas also allow for overriding local physics properties. We'll explore each of these functions below.

Area properties

Areas have many properties you can use to customize their behavior.



Area2D

Monitoring ☑ On

Monitorable ☑ On

Priority 0

Gravity

Space Override Disabled

Linear Damp

Space Override Disabled

Angular Damp

Space Override Disabled

Audio Bus

Override ☐ On

Name Master

CollisionObject2D

Disable Mode Remove

Collision

Layer

1	2	3	4	9	10	11	12
5	6	7	8	13	14	15	16

Mask

1	2	3	4	9	10	11	12
5	6	7	8	13	14	15	16

Priority 1

Input

Pickable ☑ On

The `Gravity`, `Linear Damp`, and `Angular Damp` sections are used to configure the area's physics override behavior. We'll look at how to use those in the *Area influence* section below.

`Monitoring` and `Monitorable` are used to enable and disable the area.

The `Audio Bus` section allows you to override audio in the area, for example to apply an audio effect when the player moves through.

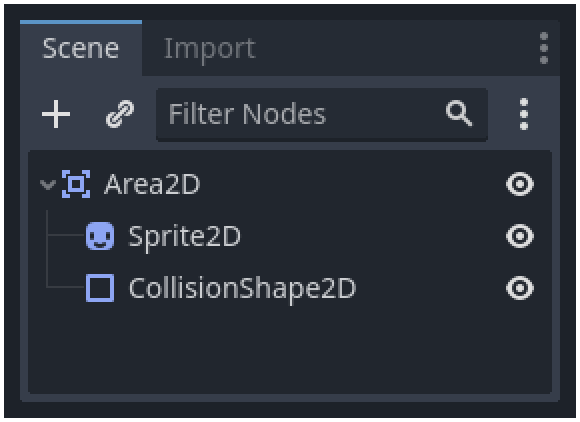
Note that Area2D extends CollisionObject2D, so it also provides properties inherited from that class. The `Collision` section of `CollisionObject2D` is where you configure the area's collision layer(s) and mask(s).

Overlap detection

Perhaps the most common use of Area2D nodes is for contact and overlap detection. When you need to know that two objects have touched, but don't need physical collision, you can use an area to notify you of the contact.

For example, let's say we're making a coin for the player to pick up. The coin is not a solid object - the player can't stand on it or push it - we just want it to disappear when the player touches it.

Here's the node setup for the coin:



To detect the overlap, we'll connect the appropriate signal on the Area2D. Which signal to use depends on the player's node type. If the player is another area, use `area_entered`. However, let's assume our player is a `CharacterBody2D` (and therefore a `CollisionObject2D` type), so we'll connect the `body_entered` signal.

Note

If you're not familiar with using signals, see Using signals for an introduction.

GDScript

```
extends Area2D

func _on_coin_body_entered(body):
    queue_free()
```

C#

```
using Godot;

public partial class Coin : Area2D
{
    private void OnCoinBodyEntered(PhysicsBody2D body)
    {
        QueueFree();
    }
}
```

Now our player can collect the coins!

Some other usage examples:

- Areas are great for bullets and other projectiles that hit and deal damage, but don't need any other physics such as bouncing.
- Use a large circular area around an enemy to define its "detect" radius. When the player is outside the area, the enemy can't "see" it.
- "Security cameras" - In a large level with multiple cameras, attach areas to each camera and activate them when the player

enters.

See the Your first 2D game for an example of using Area2D in a game.

Area influence

The second major use for area nodes is to alter physics. By default, the area won't do this, but you can enable this with the `Space Override` property. When areas overlap, they are processed in `Priority` order (higher priority areas are processed first). There are four options for override:

- *Combine* - The area adds its values to what has been calculated so far.
- *Replace* - The area replaces physics properties, and lower priority areas are ignored.
- *Combine-Replace* - The area adds its gravity/damping values to whatever has been calculated so far (in priority order), ignoring any lower priority areas.
- *Replace-Combine* - The area replaces any gravity/damping calculated so far, but keeps calculating the rest of the areas.

Using these properties, you can create very complex behavior with multiple overlapping areas.

The physics properties that can be overridden are:

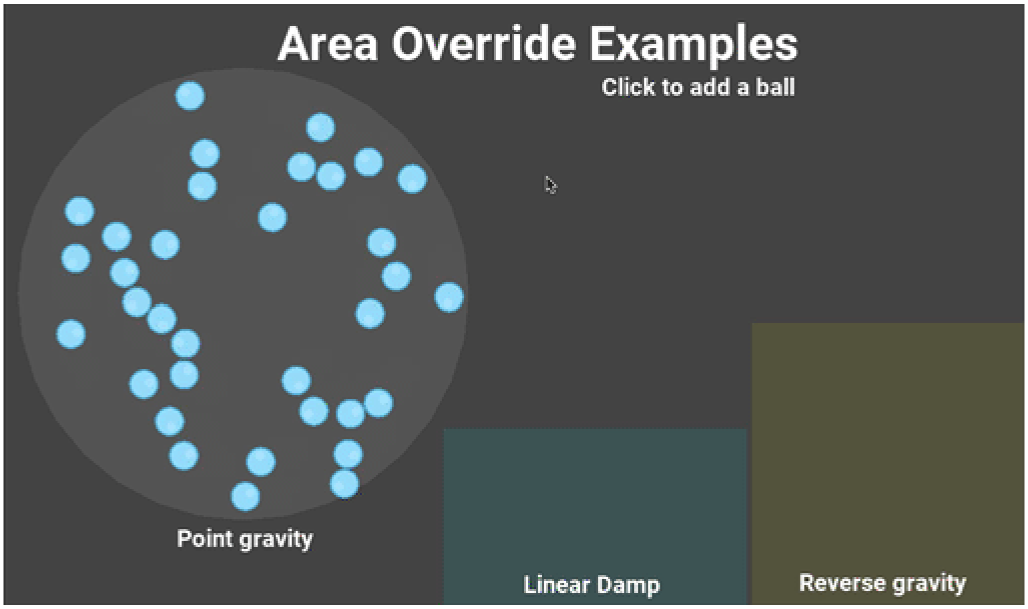
- *Gravity* - Gravity's strength inside the area.
- *Gravity Direction* - This vector does not need to be normalized.
- *Linear Damp* - How quickly objects stop moving - linear velocity lost per second.
- *Angular Damp* - How quickly objects stop spinning - angular velocity lost per second.

Point gravity

The `Gravity Point` property allows you to create an "attractor". Gravity in the area will be calculated towards a point, given by the `Point Center` property. Values are relative to the Area2D, so for example using `(0, 0)` will attract objects to the center of the area.

Examples

The example project attached below has three areas demonstrating physics override.


Area Override Examples
Click to add a ball
Point gravity
Linear Damp
Reverse gravity

You can download this project here: [area_2d_starter.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/area_2d_starter.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/area_2d_starter.zip]

Using CharacterBody2D/3D

Introduction

Godot offers several collision objects to provide both collision detection and response. Trying to decide which one to use for your project can be confusing. You can avoid problems and simplify development if you understand how each of them works and what their pros and cons are. In this tutorial, we'll look at the CharacterBody2D node and show some examples of how to use it.

Note

While this document uses `CharacterBody2D` in its examples, the same concepts apply in 3D as well.

What is a character body?

`CharacterBody2D` is for implementing bodies that are controlled via code. Character bodies detect collisions with other bodies when moving, but are not affected by engine physics properties, like gravity or friction. While this means that you have to write some code to create their behavior, it also means you have more precise control over how they move and react.

Note

This document assumes you're familiar with Godot's various physics bodies. Please read Physics introduction first, for an overview of the physics options.

Tip

A *CharacterBody2D* can be affected by gravity and other forces, but you must calculate the movement in code. The physics engine will not move a *CharacterBody2D*.

Movement and collision

When moving a `CharacterBody2D`, you should not set its `position` property directly. Instead, you use the `move_and_collide()` or `move_and_slide()` methods. These methods move the body along a given vector and detect collisions.

Warning

You should handle physics body movement in the `_physics_process()` callback.

The two movement methods serve different purposes, and later in this tutorial, you'll see examples of how they work.

move_and_collide

This method takes one required parameter: a Vector2 indicating the body's relative movement. Typically, this is your velocity vector multiplied by the frame timestep (`delta`). If the engine detects a collision anywhere along this vector, the body will immediately stop moving. If this happens, the method will return a [KinematicCollision2D]() object.

`KinematicCollision2D` is an object containing data about the collision and the colliding object. Using this data, you can calculate your collision response.

`move_and_collide` is most useful when you just want to move the body and detect collision, but don't need any automatic collision response. For example, if you need a bullet that ricochets off a wall, you can directly change the angle of the velocity when you detect a collision. See below for an example.

move_and_slide

The `move_and_slide()` method is intended to simplify the collision response in the common case where you want one body to slide along the other. It is especially useful in platformers or top-down games, for example.

When calling `move_and_slide()`, the function uses a number of node properties to calculate its slide behavior. These properties can be found in the Inspector, or set in code.

- `velocity` - *default value:* `Vector2( 0, 0 )`

  This property represents the body's velocity vector in pixels per second. `move_and_slide()` will modify this value automatically when colliding.

- `motion_mode` - *default value:* `MOTION_MODE_GROUNDED`

  This property is typically used to distinguish between side-scrolling and top-down movement. When using the default value, you can use the `is_on_floor()`, `is_on_wall()`, and `is_on_ceiling()` methods to detect what type of surface the body is in contact with, and the body will interact with slopes. When using `MOTION_MODE_FLOATING`, all collisions will be considered "walls".

- `up_direction` - *default value:* `Vector2( 0, -1 )`

  This property allows you to define what surfaces the engine should consider being the floor. Its value lets you use the `is_on_floor()`, `is_on_wall()`, and `is_on_ceiling()` methods to detect what type of surface the body is in contact with. The default value means that the top side of horizontal surfaces will be considered "ground".

- `floor_stop_on_slope` - *default value:* `true`

  This parameter prevents a body from sliding down slopes when standing still.

- `wall_min_slide_angle` - *default value:* `0.261799` (in radians, equivalent to `15` degrees)

  This is the minimum angle where the body is allowed to slide when it hits a slope.

- `floor_max_angle` - *default value:* `0.785398` (in radians, equivalent to `45` degrees)

  This parameter is the maximum angle before a surface is no longer considered a "floor."

There are many other properties that can be used to modify the body's behavior under specific circumstances. See the CharacterBody2D docs for full details.

Detecting collisions

When using `move_and_collide()` the function returns a `KinematicCollision2D` directly, and you can use this in your code.

When using `move_and_slide()` it's possible to have multiple collisions occur, as the slide response is calculated. To process these collisions, use `get_slide_collision_count()` and `get_slide_collision()`:

GDScript

```
# Using move_and_collide.
var collision = move_and_collide(velocity * delta)
if collision:
    print("I collided with ", collision.get_collider().name)

# Using move_and_slide.
move_and_slide()
for i in get_slide_collision_count():
    var collision = get_slide_collision(i)
    print("I collided with ", collision.get_collider().name)
```

C#

```
// Using MoveAndCollide.
var collision = MoveAndCollide(Velocity * (float)delta);
if (collision != null)
{
    GD.Print("I collided with ",
((Node)collision.GetCollider()).Name);
}

// Using MoveAndSlide.
MoveAndSlide();
for (int i = 0; i < GetSlideCollisionCount(); i++)
{
    var collision = GetSlideCollision(i);
    GD.Print("I collided with ",
((Node)collision.GetCollider()).Name);
}
```

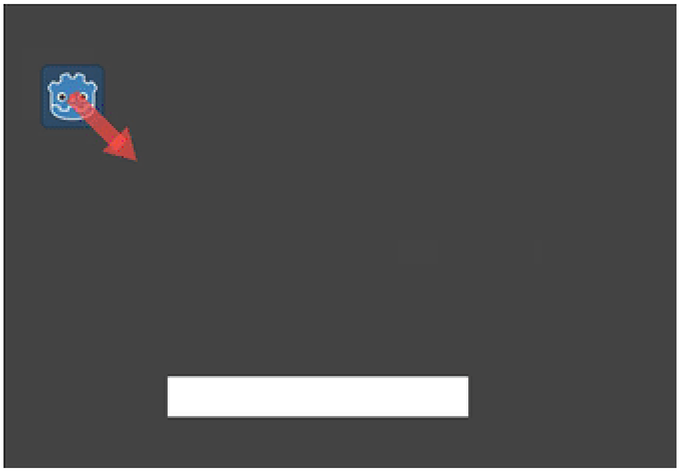
Note

get_slide_collision_count() only counts times the body has collided and changed direction.

See KinematicCollision2D for details on what collision data is returned.

Which movement method to use?

A common question from new Godot users is: "How do you decide which movement function to use?" Often, the response is to use `move_and_slide()` because it seems simpler, but this is not necessarily the case. One way to think of it is that `move_and_slide()` is a special case, and `move_and_collide()` is more general. For example, the following two code snippets result in the same collision response:



GDScript

```
# using move_and_collide
var collision = move_and_collide(velocity * delta)
if collision:
    velocity = velocity.slide(collision.get_normal())
```

```
# using move_and_slide
move_and_slide()
```

C#

```
// using MoveAndCollide
var collision = MoveAndCollide(Velocity * (float)delta);
if (collision != null)
{
    Velocity = Velocity.Slide(collision.GetNormal());
}

// using MoveAndSlide
MoveAndSlide();
```

Anything you do with `move_and_slide()` can also be done with `move_and_collide()`, but it might take a little more code. However, as we'll see in the examples below, there are cases where `move_and_slide()` doesn't provide the response you want.

In the example above, `move_and_slide()` automatically alters the `velocity` variable. This is because when the character collides with the environment, the function recalculates the speed internally to reflect the slowdown.

For example, if your character fell on the floor, you don't want it to accumulate vertical speed due to the effect of gravity. Instead, you want its vertical speed to reset to zero.

`move_and_slide()` may also recalculate the kinematic body's velocity several times in a loop as, to produce a smooth motion, it moves the character and collides up to five times by default. At the end of the process, the character's new velocity is available for use on the next frame.

Examples

To see these examples in action, download the sample project: character_body_2d_starter.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/character_body_2d_starter.zip]

Movement and walls

If you've downloaded the sample project, this example is in "basic_movement.tscn".

For this example, add a `CharacterBody2D` with two children: a `Sprite2D` and a `CollisionShape2D`. Use the Godot "icon.svg" as the Sprite2D's texture (drag it from the Filesystem dock to the *Texture* property of the `Sprite2D`). In the `CollisionShape2D`'s *Shape* property, select "New RectangleShape2D" and size the rectangle to fit over the sprite image.

Note

See 2D movement overview for examples of implementing 2D movement schemes.

Attach a script to the CharacterBody2D and add the following code:

GDScript

```
extends CharacterBody2D

var speed = 300

func get_input():
    var input_dir = Input.get_vector("ui_left", "ui_right",
"ui_up", "ui_down")
    velocity = input_dir * speed

func _physics_process(delta):
    get_input()
    move_and_collide(velocity * delta)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private int _speed = 300;

    public void GetInput()
    {
        Vector2 inputDir = Input.GetVector("ui_left",
"ui_right", "ui_up", "ui_down");
        Velocity = inputDir * _speed;
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        MoveAndCollide(Velocity * (float)delta);
    }
}
```

Run this scene and you'll see that `move_and_collide()` works as expected, moving the body along the velocity vector. Now let's see what happens when you add some obstacles. Add a StaticBody2D with a rectangular collision shape. For visibility, you can use a Sprite2D, a Polygon2D, or turn on "Visible Collision Shapes" from the "Debug" menu.

Run the scene again and try moving into the obstacle. You'll see that the `CharacterBody2D` can't penetrate the obstacle. However, try moving into the obstacle at an angle and you'll find that the obstacle acts like glue - it feels like the body gets stuck.

This happens because there is no *collision response*. `move_and_collide()` stops the body's movement when a collision occurs. We need to code whatever response we want from the collision.

Try changing the function to `move_and_slide()` and running again.

`move_and_slide()` provides a default collision response of sliding the body along the collision object. This is useful for a great many game types, and may be all you need to get the behavior you want.

Bouncing/reflecting

What if you don't want a sliding collision response? For this example ("bounce_and_collide.tscn" in the sample project), we have a character shooting bullets and we want the bullets to bounce off the walls.

This example uses three scenes. The main scene contains the Player and Walls. The Bullet and Wall are separate scenes so that they can be instanced.

The Player is controlled by the `w` and `s` keys for forward and back. Aiming uses the mouse pointer. Here is the code for the Player, using `move_and_slide()`:

GDScript

```
extends CharacterBody2D

var Bullet = preload("res://bullet.tscn")
var speed = 200

func get_input():
    # Add these actions in Project Settings -> Input Map.
    var input_dir = Input.get_axis("backward", "forward")
    velocity = transform.x * input_dir * speed
    if Input.is_action_just_pressed("shoot"):
        shoot()
```

```
func shoot():
    # "Muzzle" is a Marker2D placed at the barrel of the gun.
    var b = Bullet.instantiate()
    b.start($Muzzle.global_position, rotation)
    get_tree().root.add_child(b)

func _physics_process(delta):
    get_input()
    var dir = get_global_mouse_position() - global_position
    # Don't move if too close to the mouse pointer.
    if dir.length() > 5:
        rotation = dir.angle()
        move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private PackedScene _bullet = GD.Load<PackedScene>
("res://Bullet.tscn");
    private int _speed = 200;

    public void GetInput()
    {
        // Add these actions in Project Settings -> Input Map.
        float inputDir = Input.GetAxis("backward", "forward");
        Velocity = Transform.X * inputDir * _speed;
        if (Input.IsActionPressed("shoot"))
        {
            Shoot();
        }
    }

    public void Shoot()
    {
        // "Muzzle" is a Marker2D placed at the barrel of the
gun.
        var b = (Bullet)_bullet.Instantiate();
        b.Start(GetNode<Node2D>("Muzzle").GlobalPosition,
Rotation);
```

```
        GetTree().Root.AddChild(b);
    }

    public override void _PhysicsProcess(double delta)
    {
        GetInput();
        var dir = GetGlobalMousePosition() - GlobalPosition;
        // Don't move if too close to the mouse pointer.
        if (dir.Length() > 5)
        {
            Rotation = dir.Angle();
            MoveAndSlide();
        }
    }
}
```

And the code for the Bullet:

GDScript

```
extends CharacterBody2D

var speed = 750

func start(_position, _direction):
    rotation = _direction
    position = _position
    velocity = Vector2(speed, 0).rotated(rotation)

func _physics_process(delta):
    var collision = move_and_collide(velocity * delta)
    if collision:
        velocity = velocity.bounce(collision.get_normal())
        if collision.get_collider().has_method("hit"):
            collision.get_collider().hit()

func _on_VisibilityNotifier2D_screen_exited():
    # Deletes the bullet when it exits the screen.
    queue_free()
```

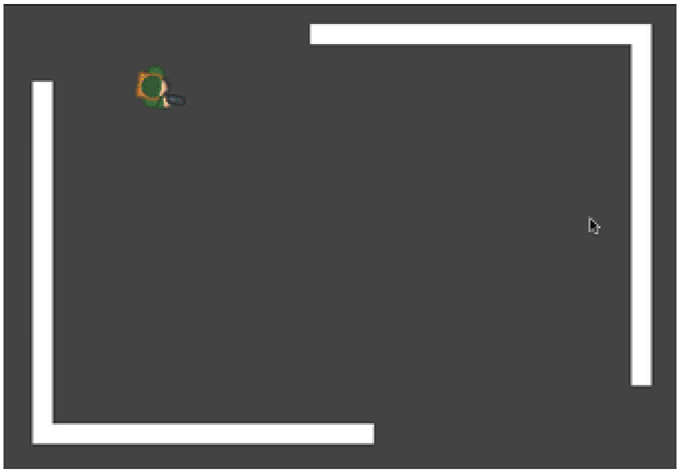
C#

```
using Godot;

public partial class Bullet : CharacterBody2D
{
    public int _speed = 750;

    public void Start(Vector2 position, float direction)
    {
        Rotation = direction;
        Position = position;
        Velocity = new Vector2(speed, 0).Rotated(Rotation);
    }

    public override void _PhysicsProcess(double delta)
    {
        var collision = MoveAndCollide(Velocity *
(float)delta);
        if (collision != null)
        {
            Velocity = Velocity.Bounce(collision.GetNormal());
            if (collision.GetCollider().HasMethod("Hit"))
            {
                collision.GetCollider().Call("Hit");
            }
        }
    }

    private void OnVisibilityNotifier2DScreenExited()
    {
        // Deletes the bullet when it exits the screen.
        QueueFree();
    }
}
```

The action happens in `_physics_process()`. After using `move_and_collide()`, if a collision occurs, a `KinematicCollision2D` object is returned (otherwise, the return is `null`).

If there is a returned collision, we use the `normal` of the collision to reflect the bullet's `velocity` with the `Vector2.bounce()` method.

If the colliding object (`collider`) has a `hit` method, we also call it. In the example project, we've added a flashing color effect to the Wall to demonstrate this.



Platformer movement

Let's try one more popular example: the 2D platformer. `move_and_slide()` is ideal for quickly getting a functional character controller up and running. If you've downloaded the sample project, you can find this in "platformer.tscn".

For this example, we'll assume you have a level made of one or more `StaticBody2D` objects. They can be any shape and size. In the sample project, we're using Polygon2D to create the platform shapes.

Here's the code for the player body:

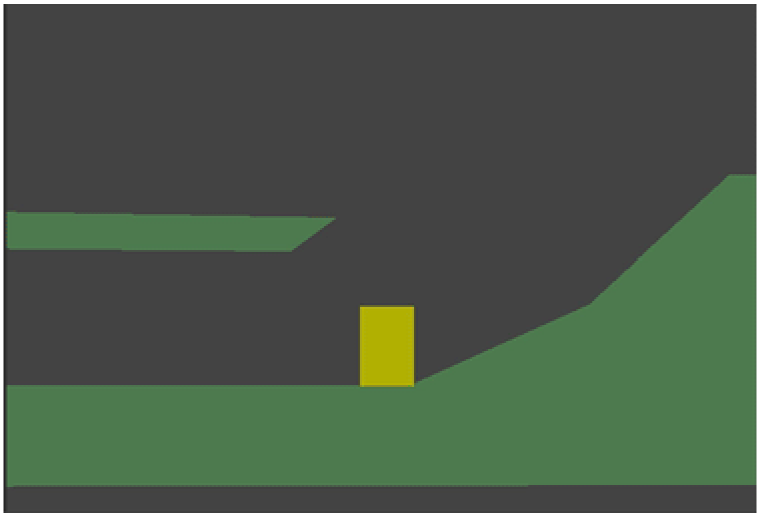
GDScript

```
extends CharacterBody2D

var speed = 300.0
var jump_speed = -400.0

# Get the gravity from the project settings so you can sync
```

```
with rigid body nodes.
var gravity =
ProjectSettings.get_setting("physics/2d/default_gravity")


func _physics_process(delta):
    # Add the gravity.
    velocity.y += gravity * delta

    # Handle Jump.
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = jump_speed

    # Get the input direction.
    var direction = Input.get_axis("ui_left", "ui_right")
    velocity.x = direction * speed

    move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private float _speed = 100.0f;
    private float _jumpSpeed = -400.0f;

    // Get the gravity from the project settings so you can
sync with rigid body nodes.
    public float Gravity =
ProjectSettings.GetSetting("physics/2d/default_gravity").AsSing
le();

    public override void _PhysicsProcess(double delta)
    {
        Vector2 velocity = Velocity;

        // Add the gravity.
        velocity.Y += Gravity * (float)delta;

        // Handle jump.
        if (Input.IsActionJustPressed("jump") && IsOnFloor())
            velocity.Y = _jumpSpeed;

        // Get the input direction.
        float direction = Input.GetAxis("ui_left", "ui_right");
        velocity.X = direction * _speed;

        Velocity = velocity;
        MoveAndSlide();
    }
}
```



In this code we're using `move_and_slide()` as described above - to move the body along its velocity vector, sliding along any collision surfaces such as the ground or a platform. We're also using `is_on_floor()` to check if a jump should be allowed. Without this, you'd be able to "jump" in midair; great if you're making Flappy Bird, but not for a platformer game.

There is a lot more that goes into a complete platformer character: acceleration, double-jumps, coyote-time, and many more. The code above is just a starting point. You can use it as a base to expand into whatever movement behavior you need for your own projects.

Ray-casting

Introduction

One of the most common tasks in game development is casting a ray (or custom shaped object) and checking what it hits. This enables complex behaviors, AI, etc. to take place. This tutorial will explain how to do this in 2D and 3D.

Godot stores all the low level game information in servers, while the scene is only a frontend. As such, ray casting is generally a lower-level task. For simple raycasts, nodes like RayCast3D and RayCast2D will work, as they return every frame what the result of a raycast is.

Many times, though, ray-casting needs to be a more interactive process so a way to do this by code must exist.

Space

In the physics world, Godot stores all the low level collision and physics information in a *space*. The current 2d space (for 2D Physics) can be obtained by accessing CanvasItem.get_world_2d().space. For 3D, it's Node3D.get_world_3d().space.

The resulting space RID can be used in PhysicsServer3D and PhysicsServer2D respectively for 3D and 2D.

Accessing space

Godot physics runs by default in the same thread as game logic, but may be set to run on a separate thread to work more efficiently. Due to this, the only time accessing space is safe is during the Node._physics_process() callback. Accessing it from outside this function may result in an error due to space being *locked*.

To perform queries into physics space, the PhysicsDirectSpaceState2D and PhysicsDirectSpaceState3D must be used.

Use the following code in 2D:

GDScript

```
func _physics_process(delta):
    var space_rid = get_world_2d().space
    var space_state =
PhysicsServer2D.space_get_direct_state(space_rid)
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceRid = GetWorld2D().Space;
    var spaceState =
PhysicsServer2D.SpaceGetDirectState(spaceRid);
}
```

Or more directly:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld2D().DirectSpaceState;
}
```

And in 3D:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_3d().direct_space_state
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld3D().DirectSpaceState;
}
```

Raycast query

For performing a 2D raycast query, the method PhysicsDirectSpaceState2D.intersect_ray() may be used. For example:

GDScript

```
func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    # use global coordinates, not local to node
    var query = PhysicsRayQueryParameters2D.create(Vector2(0,
0), Vector2(50, 100))
    var result = space_state.intersect_ray(query)
```

C#

```
public override void _PhysicsProcess(double delta)
{
    var spaceState = GetWorld2D().DirectSpaceState;
    // use global coordinates, not local to node
    var query =
PhysicsRayQueryParameters2D.Create(Vector2.Zero, new
Vector2(50, 100));
    var result = spaceState.IntersectRay(query);
}
```

The result is a dictionary. If the ray didn't hit anything, the dictionary will be empty. If it did hit something, it will contain collision information:

GDScript

```
if result:
    print("Hit at point: ", result.position)
```

C#

```
if (result.Count > 0)
    GD.Print("Hit at point: ", result["position"]);
```

The `result` dictionary when a collision occurs contains the following data:

```
{
   position: Vector2 # point in world space for collision
   normal: Vector2 # normal in world space for collision
   collider: Object # Object collided or null (if unassociated)
   collider_id: ObjectID # Object it collided against
   rid: RID # RID it collided against
   shape: int # shape index of collider
   metadata: Variant() # metadata of collider
}
```

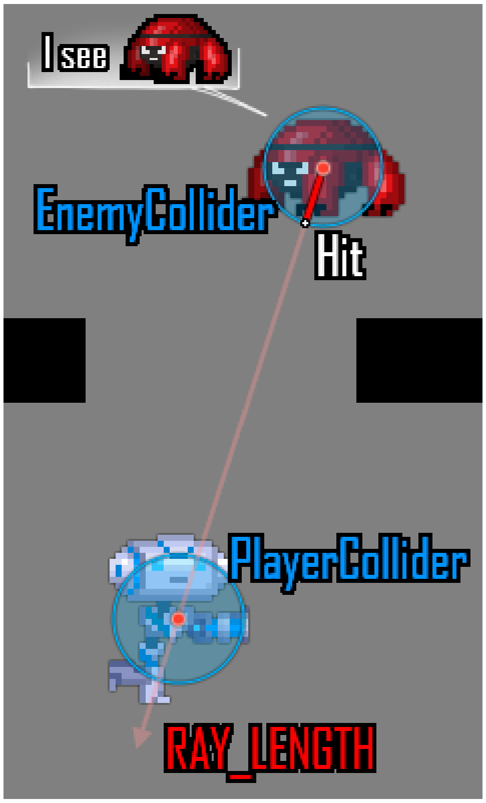
The data is similar in 3D space, using Vector3 coordinates. Note that to enable collisions with Area3D, the boolean parameter `collide_with_areas` must be set to `true`.

GDScript

```
const RAY_LENGTH = 1000

func _physics_process(delta):
    var space_state = get_world_3d().direct_space_state
    var cam = $Camera3D
    var mousepos = get_viewport().get_mouse_position()

    var origin = cam.project_ray_origin(mousepos)
    var end = origin + cam.project_ray_normal(mousepos) *
RAY_LENGTH
    var query = PhysicsRayQueryParameters3D.create(origin, end)
    query.collide_with_areas = true

    var result = space_state.intersect_ray(query)
```

Collision exceptions

A common use case for ray casting is to enable a character to gather data about the world around it. One problem with this is that the same character has a collider, so the ray will only detect its parent's collider, as shown in the following image:



To avoid self-intersection, the `intersect_ray()` parameters object can take an array of exceptions via its `exclude` property. This is an example of how to use it from a CharacterBody2D or any other collision object node:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    var query =
PhysicsRayQueryParameters2D.create(global_position,
player_position)
    query.exclude = [self]
    var result = space_state.intersect_ray(query)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        var spaceState = GetWorld2D().DirectSpaceState;
        var query =
PhysicsRayQueryParameters2D.Create(globalPosition,
playerPosition);
        query.Exclude = new Godot.Collections.Array<Rid> {
GetRid() };
        var result = spaceState.IntersectRay(query);
    }
}
```

The exceptions array can contain objects or RIDs.

Collision Mask

While the exceptions method works fine for excluding the parent body, it becomes very inconvenient if you need a large and/or dynamic list of exceptions. In this case, it is much more efficient to use the collision layer/mask system.

The `intersect_ray()` parameters object can also be supplied a collision mask. For example, to use the same mask as the parent body, use the `collision_mask` member variable. The array of exceptions can be supplied as the last argument as well:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    var space_state = get_world_2d().direct_space_state
    var query =
PhysicsRayQueryParameters2D.create(global_position,
target_position,
        collision_mask, [self])
    var result = space_state.intersect_ray(query)
```

C#

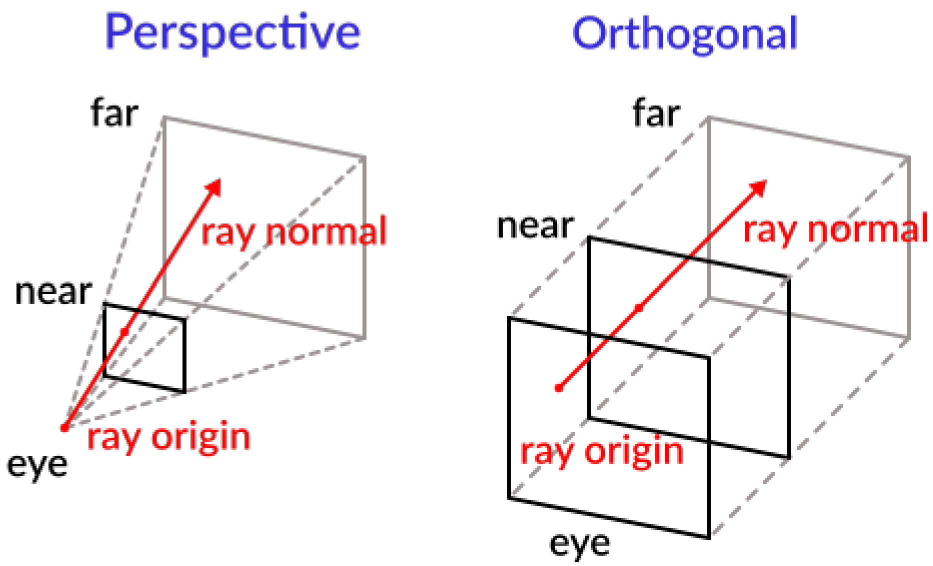
```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        var spaceState = GetWorld2D().DirectSpaceState;
        var query =
PhysicsRayQueryParameters2D.Create(globalPosition,
targetPosition,
            CollisionMask, new Godot.Collections.Array<Rid> {
GetRid() });
        var result = spaceState.IntersectRay(query);
    }
}
```

See Code example for details on how to set the collision mask.

3D ray casting from screen

Casting a ray from screen to 3D physics space is useful for object picking. There is not much need to do this because CollisionObject3D has an "input_event" signal that will let you know when it was clicked, but in case there is any desire to do it manually, here's how.

To cast a ray from the screen, you need a Camera3D node. A `Camera3D` can be in two projection modes: perspective and orthogonal. Because of this, both the ray origin and direction must be obtained. This is because `origin` changes in orthogonal mode, while `normal` changes in perspective mode:


Perspective
far
ray normal
near
ray origin
eye
Orthogonal
far
near
ray normal
ray origin
eye

To obtain it using a camera, the following code can be used:

GDScript

```
const RAY_LENGTH = 1000.0

func _input(event):
    if event is InputEventMouseButton and event.pressed and
event.button_index == 1:
          var camera3d = $Camera3D
          var from =
camera3d.project_ray_origin(event.position)
          var to = from +
camera3d.project_ray_normal(event.position) * RAY_LENGTH
```

C#

```
private const float RayLength = 1000.0f;

public override void _Input(InputEvent @event)
{
    if (@event is InputEventMouseButton eventMouseButton &&
eventMouseButton.Pressed && eventMouseButton.ButtonIndex ==
MouseButton.Left)
    {
        var camera3D = GetNode<Camera3D>("Camera3D");
        var from =
camera3D.ProjectRayOrigin(eventMouseButton.Position);
        var to = from +
camera3D.ProjectRayNormal(eventMouseButton.Position) *
RayLength;
    }
}
```

Remember that during `_input()`, the space may be locked, so in practice this query should be run in `_physics_process()`.

Ragdoll system

Introduction

Since version 3.1, Godot supports ragdoll physics. Ragdolls rely on physics simulation to create realistic procedural animation. They are used for death animations in many games.

In this tutorial, we will be using the Platformer3D demo to set up a ragdoll.

Note

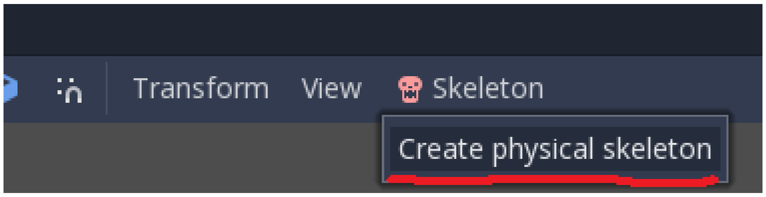
You can download the Platformer3D demo on GitHub [https://github.com/godotengine/godot-demo-projects/tree/master/3d/platformer] or using the Asset Library [https://godotengine.org/asset-library/asset/2748].

Setting up the ragdoll

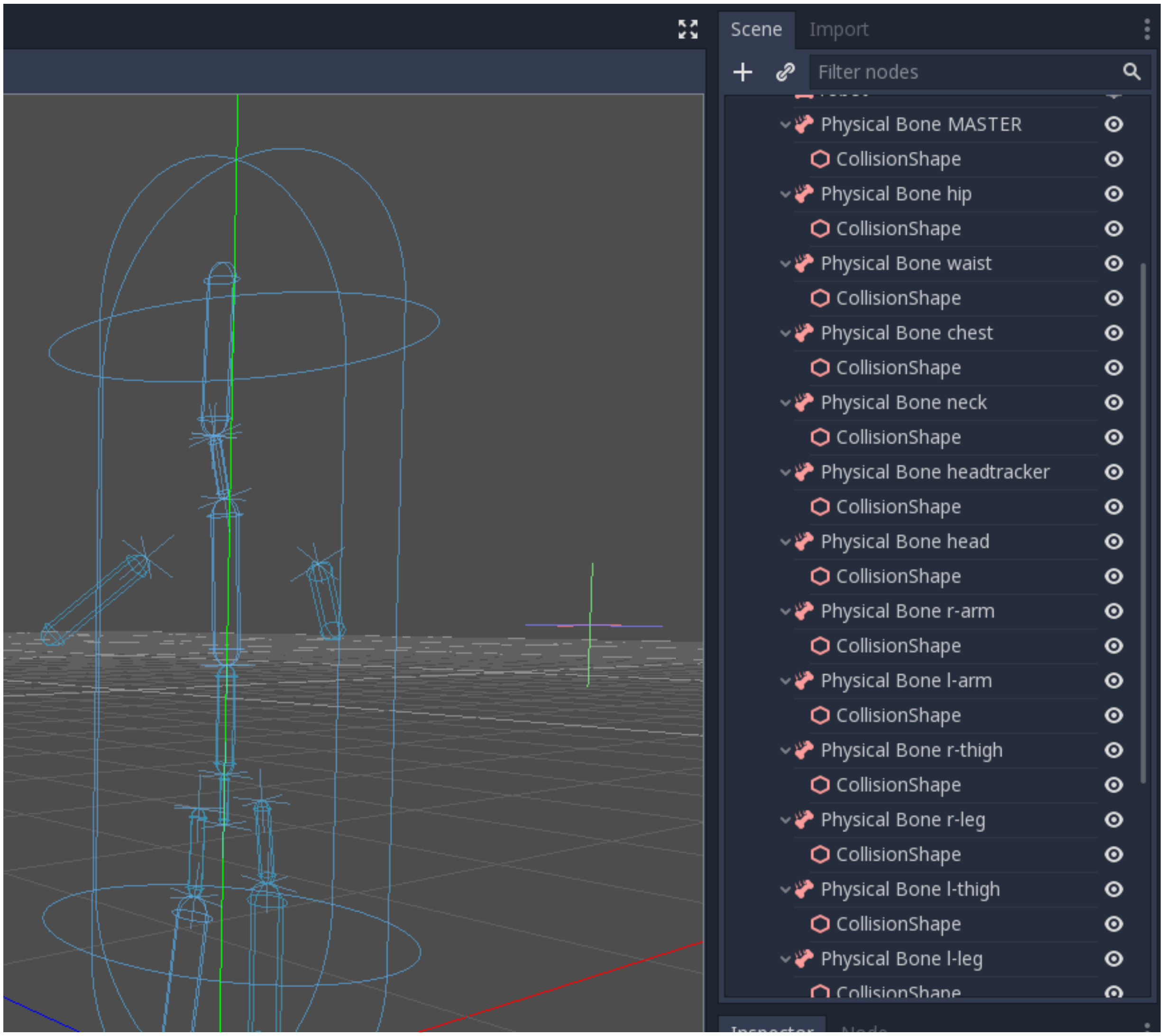
Creating physical bones

Like many other features in the engine, there is a node to set up a ragdoll: the PhysicalBone3D node. To simplify the setup, you can generate `PhysicalBone` nodes with the "Create physical skeleton" feature in the skeleton node.

Open the platformer demo in Godot, and then the Robi scene. Select the `Skeleton` node. A skeleton button appears on the top bar menu:



Click it and select the `Create physical skeleton` option. Godot will generate `PhysicalBone` nodes and collision shapes for each bone in the skeleton and pin joints to connect them together:



Physical Bone MASTER
  CollisionShape
Physical Bone hip
  CollisionShape
Physical Bone waist
  CollisionShape
Physical Bone chest
  CollisionShape
Physical Bone neck
  CollisionShape
Physical Bone headtracker
  CollisionShape
Physical Bone head
  CollisionShape
Physical Bone r-arm
  CollisionShape
Physical Bone l-arm
  CollisionShape
Physical Bone r-thigh
  CollisionShape
Physical Bone r-leg
  CollisionShape
Physical Bone l-thigh
  CollisionShape
Physical Bone l-leg
  CollisionShape

Some of the generated bones aren't necessary: the `MASTER` bone for example. So we're going to clean up the skeleton by removing them.

Cleaning up the skeleton

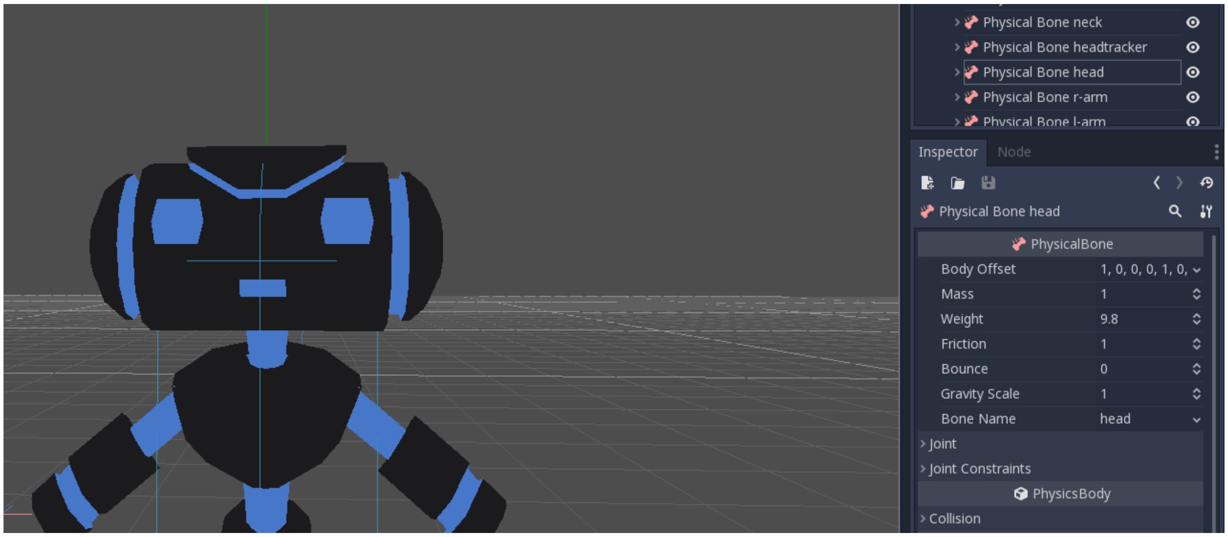
Each `PhysicalBone` the engine needs to simulate has a performance cost, so you want to remove every bone that is too small to make a difference in the simulation, as well as all utility bones.

For example, if we take a humanoid, you do not want to have physical bones for each finger. You can use a single bone for the entire hand instead, or one for the palm, one for the thumb, and a last one for the other four fingers.

Remove these physical bones: `MASTER`, `waist`, `neck`, `headtracker`. This gives us an optimized skeleton and makes it easier to control the ragdoll.

Collision shape adjustment

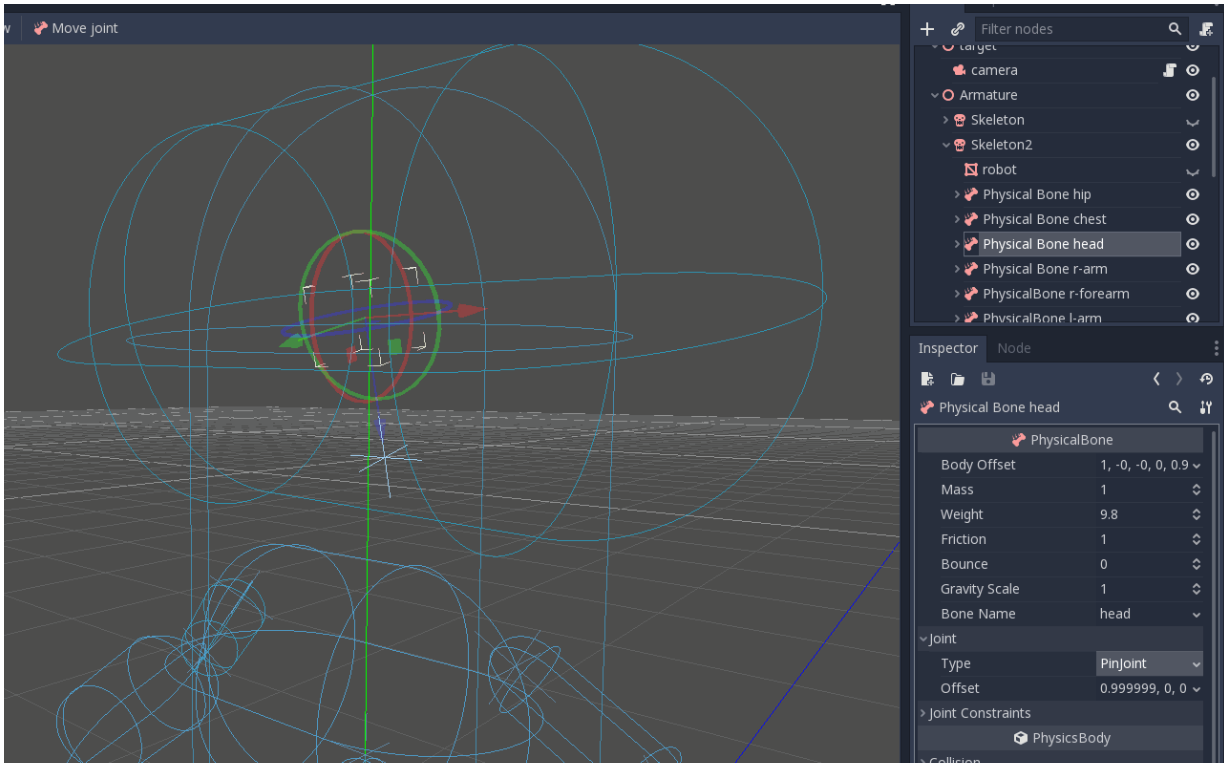
The next task is adjusting the collision shape and the size of physical bones to match the part of the body that each bone should simulate.


Physical Bone neck
Physical Bone headtracker
Physical Bone head
Physical Bone r-arm
Physical Bone l-arm
Inspector
Node
Physical Bone head
PhysicalBone
Joint
Joint Constraints
PhysicsBody
Collision

Body Offset	1, 0, 0, 0, 1, 0,
Mass	1
Weight	9.8
Friction	1
Bounce	0
Gravity Scale	1
Bone Name	head

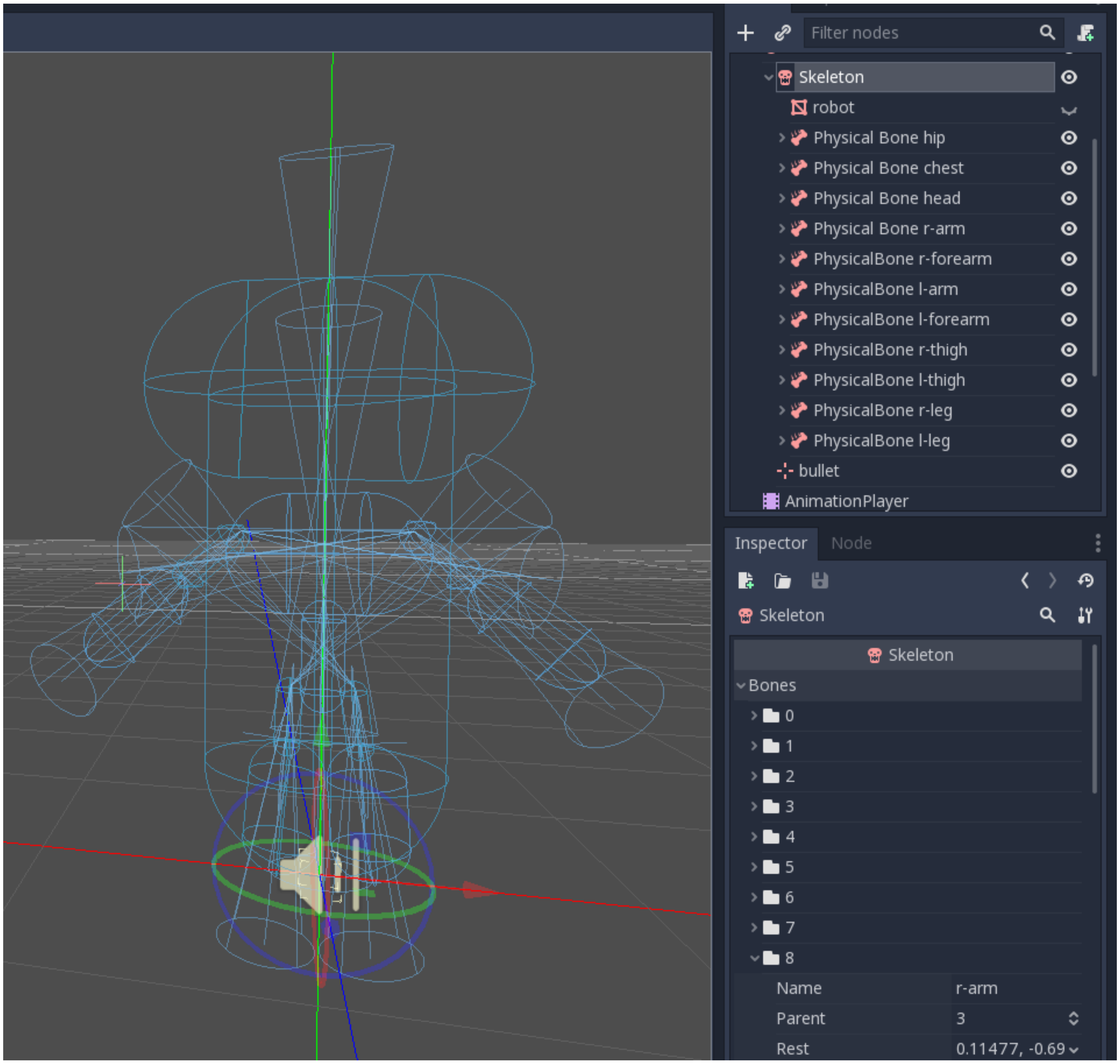
Joints adjustment

Once you adjusted the collision shapes, your ragdoll is almost ready. You just want to adjust the pin joints to get a better simulation. `PhysicalBone` nodes have an unconstrained pin joint assigned to them by default. To change the pin joint, select the `PhysicalBone` and change the constraint type in the `Joint` section. There, you can change the constraint's orientation and its limits.


Move joint
Filter nodes
camera
Armature
Skeleton
Skeleton2
robot
Physical Bone hip
Physical Bone chest
Physical Bone head
Physical Bone r-arm
PhysicalBone r-forearm
PhysicalBone l-arm
Inspector
Node
Physical Bone head
PhysicalBone
Joint
Joint Constraints
PhysicsBody

Body Offset 1, -0, -0, 0, 0.9
Mass 1
Weight 9.8
Friction 1
Bounce 0
Gravity Scale 1
Bone Name head
Type PinJoint
Offset 0.999999, 0, 0

This is the final result:


Filter nodes
Skeleton
robot
Physical Bone hip
Physical Bone chest
Physical Bone head
Physical Bone r-arm
PhysicalBone r-forearm
PhysicalBone l-arm
PhysicalBone l-forearm
PhysicalBone r-thigh
PhysicalBone l-thigh
PhysicalBone r-leg
PhysicalBone l-leg
bullet
AnimationPlayer
Inspector
Node
Skeleton
Skeleton
Bones
0
1
2
3
4
5
6
7
8

Name r-arm
Parent 3
Rest 0.11477, -0.69

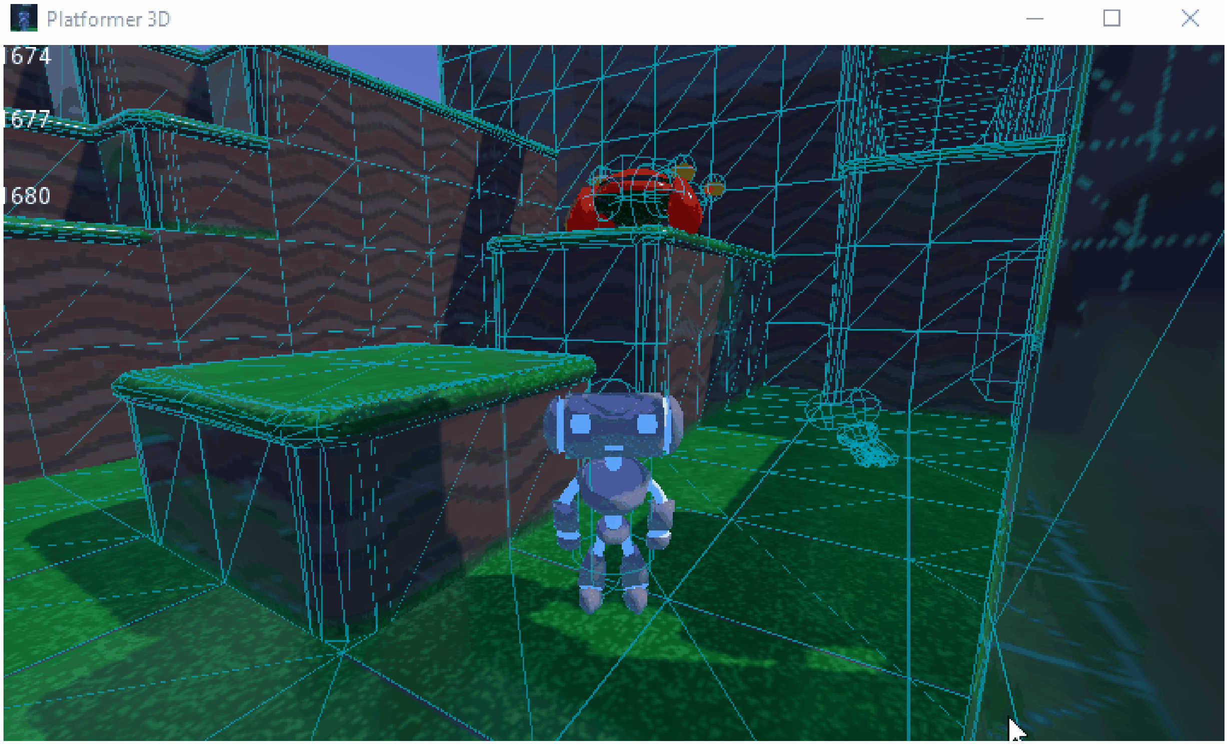
Simulating the ragdoll

The ragdoll is now ready to use. To start the simulation and play the ragdoll animation, you need to call the `physical_bones_start_simulation` method. Attach a script to the skeleton node and call the method in the `_ready` method:

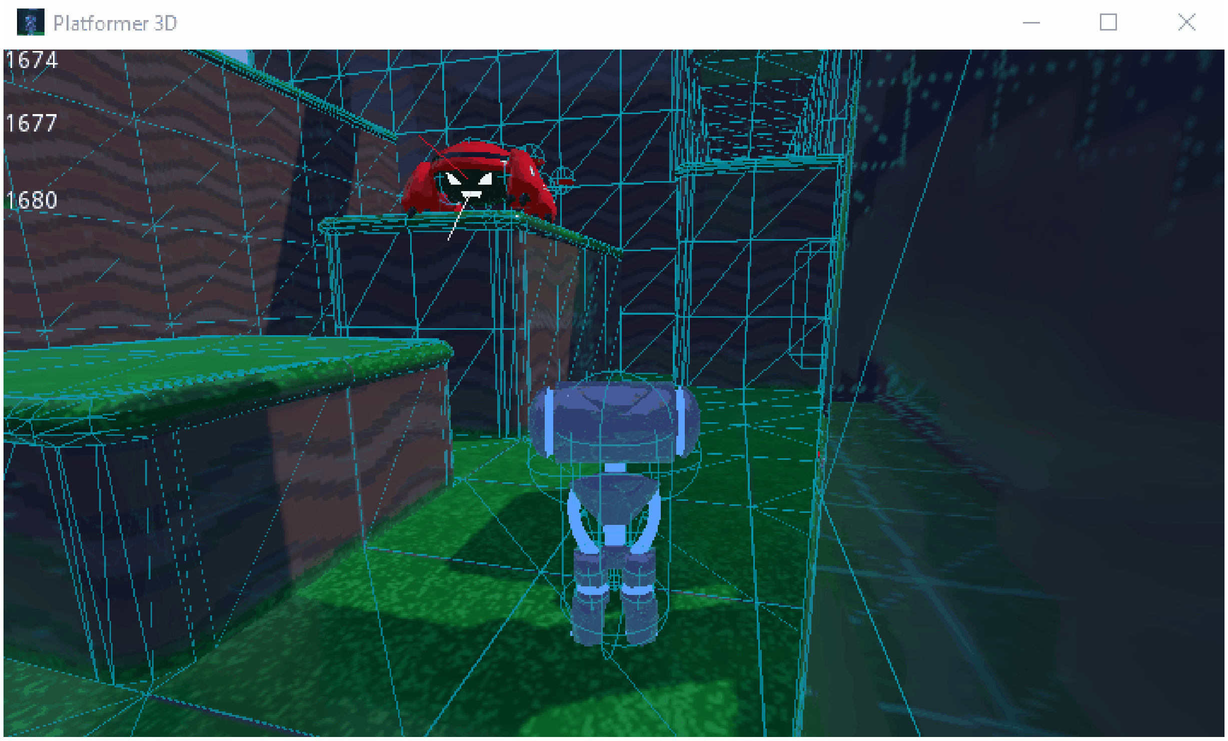
GDScript

```
func _ready():
    physical_bones_start_simulation()
```

To stop the simulation, call the `physical_bones_stop_simulation()` method.

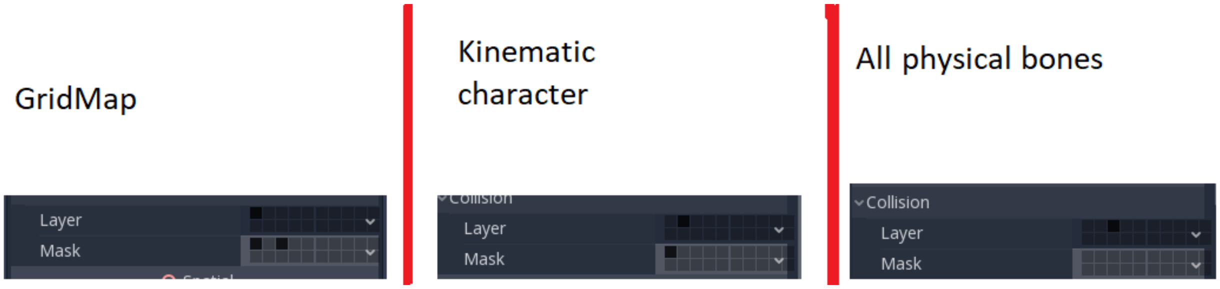

Platformer 3D
1674
1677
1680

You can also limit the simulation to only a few bones. To do so, pass the bone names as a parameter. Here's an example of partial ragdoll simulation:


Platformer 3D
1674
1677
1680

Collision layer and mask

Make sure to set up your collision layers and masks properly so the `CharacterBody3D`'s capsule doesn't get in the way of the physics simulation:


GridMap
Layer
Mask
Kinematic character
Collision
Layer
Mask
All physical bones
Collision
Layer
Mask

For more information, read [Collision layers and masks](#).

Kinematic character (2D)

Introduction

Yes, the name sounds strange. "Kinematic Character". What is that? The reason for the name is that, when physics engines came out, they were called "Dynamics" engines (because they dealt mainly with collision responses). Many attempts were made to create a character controller using the dynamics engines, but it wasn't as easy as it seemed. Godot has one of the best implementations of dynamic character controller you can find (as it can be seen in the 2d/platformer demo), but using it requires a considerable level of skill and understanding of physics engines (or a lot of patience with trial and error).

Some physics engines, such as Havok seem to swear by dynamic character controllers as the best option, while others (PhysX) would rather promote the kinematic one.

So, what is the difference?:

- A **dynamic character controller** uses a rigid body with an infinite inertia tensor. It's a rigid body that can't rotate. Physics engines always let objects move and collide, then solve their collisions all together. This makes dynamic character controllers able to interact with other physics objects seamlessly, as seen in the platformer demo. However, these interactions are not always predictable. Collisions can take more than one frame to be solved, so a few collisions may seem to displace a tiny bit. Those problems can be fixed, but require a certain amount of skill.
- A **kinematic character controller** is assumed to always begin in a non-colliding state, and will always move to a non-colliding

> state. If it starts in a colliding state, it will try to free itself like rigid bodies do, but this is the exception, not the rule. This makes their control and motion a lot more predictable and easier to program. However, as a downside, they can't directly interact with other physics objects, unless done by hand in code.

This short tutorial focuses on the kinematic character controller. It uses the old-school way of handling collisions, which is not necessarily simpler under the hood, but well hidden and presented as an API.

Physics process

To manage the logic of a kinematic body or character, it is always advised to use physics process, because it's called before physics step and its execution is in sync with physics server, also it is called the same amount of times per second, always. This makes physics and motion calculation work in a more predictable way than using regular process, which might have spikes or lose precision if the frame rate is too high or too low.

GDScript

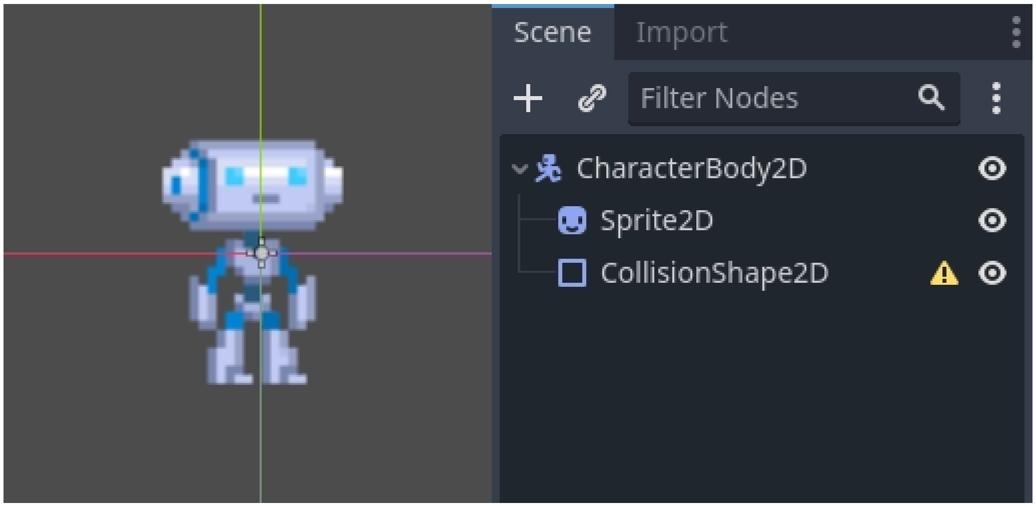
```
extends CharacterBody2D

func _physics_process(delta):
    pass
```

C#

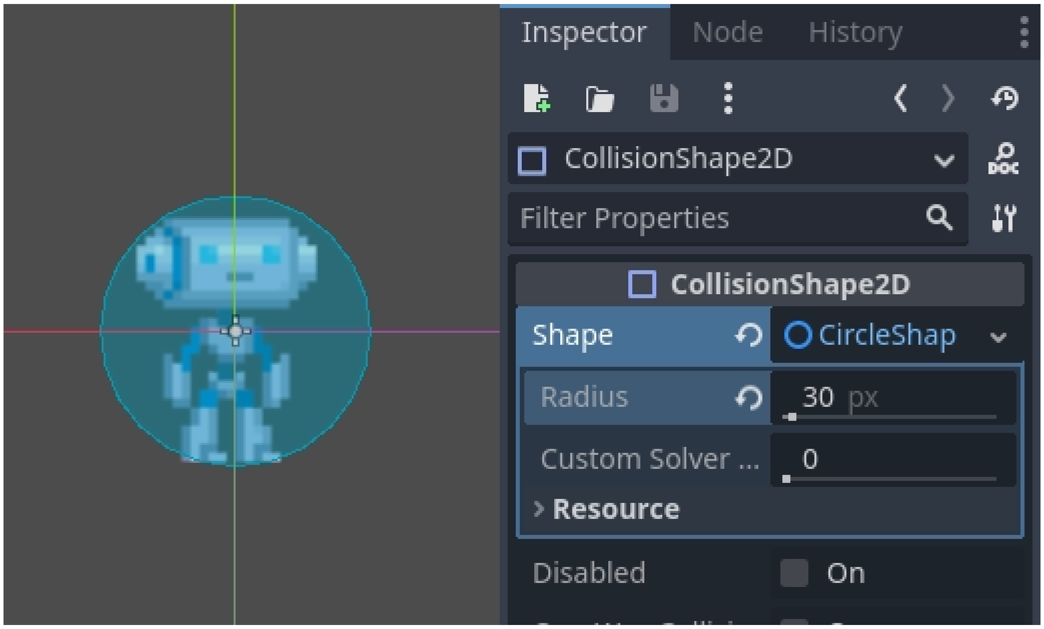
```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
    }
}
```

Scene setup

To have something to test, here's the scene (from the tilemap tutorial): [kinematic_character_2d_starter.zip](https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/kinematic_character_2d_starter.zip) [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/kinematic_character_2d_starter.zip]. We'll be creating a new scene for the character. Use the robot sprite and create a scene like this:



You'll notice that there's a warning icon next to our CollisionShape2D node; that's because we haven't defined a shape for it. Create a new CircleShape2D in the shape property of CollisionShape2D. Click on <CircleShape2D> to go to the options for it, and set the radius to 30:


Inspector
Node
History
CollisionShape2D
Filter Properties
CollisionShape2D
Resource

Shape: CircleShap

Radius: 30 px

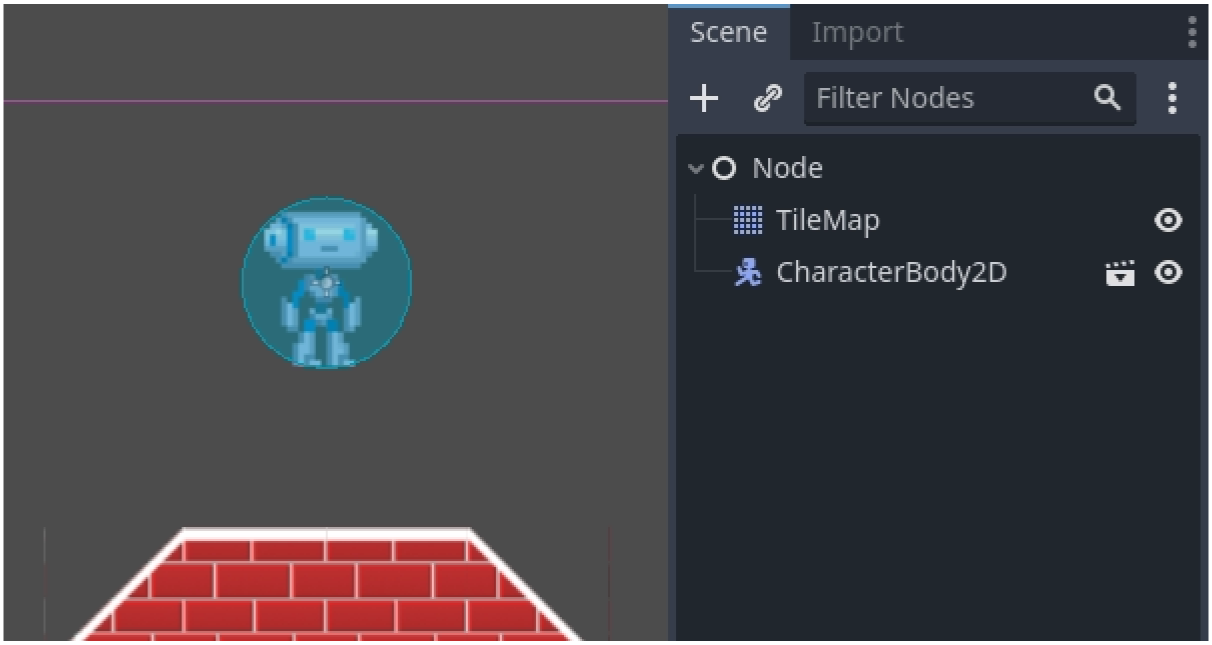
Custom Solver ...: 0

Disabled: ☐ On

Note: As mentioned before in the physics tutorial, the physics engine can't handle scale on most types of shapes (only collision polygons, planes and segments work), so always change the parameters (such as radius) of the shape instead of scaling it. The same is also true for the kinematic/rigid/static bodies themselves, as their scale affects the shape scale.

Now, create a script for the character, the one used as an example above should work as a base.

Finally, instance that character scene in the tilemap, and make the map scene the main one, so it runs when pressing play.



Moving the kinematic character

Go back to the character scene, and open the script, the magic begins now! Kinematic body will do nothing by default, but it has a useful function called `CharacterBody2D.move_and_collide()`. This function takes a Vector2 as an argument, and tries to apply that motion to the kinematic body. If a collision happens, it stops right at the moment of the collision.

So, let's move our sprite downwards until it hits the floor:

GDScript

```
extends CharacterBody2D

func _physics_process(delta):
    move_and_collide(Vector2(0, 1)) # Move down 1 pixel per physics frame
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    public override void _PhysicsProcess(double delta)
    {
        // Move down 1 pixel per physics frame
        MoveAndCollide(new Vector2(0, 1));
    }
}
```

The result is that the character will move, but stop right when hitting the floor. Pretty cool, huh?

The next step will be adding gravity to the mix, this way it behaves a little more like a regular game character:

GDScript

```
extends CharacterBody2D

const GRAVITY = 200.0

func _physics_process(delta):
    velocity.y += delta * GRAVITY

    var motion = velocity * delta
    move_and_collide(motion)
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private const float Gravity = 200.0f;

    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;
        velocity.Y += (float)delta * Gravity;
        Velocity = velocity;
        var motion = velocity * (float)delta;
        MoveAndCollide(motion);
    }
}
```

Now the character falls smoothly. Let's make it walk to the sides, left and right when touching the directional keys. Remember that the values being used (for speed at least) are pixels/second.

This adds basic support for walking when pressing left and right:

GDScript

```
extends CharacterBody2D

const GRAVITY = 200.0
const WALK_SPEED = 200

func _physics_process(delta):
    velocity.y += delta * GRAVITY

    if Input.is_action_pressed("ui_left"):
        velocity.x = -WALK_SPEED
    elif Input.is_action_pressed("ui_right"):
        velocity.x =  WALK_SPEED
    else:
        velocity.x = 0

    # "move_and_slide" already takes delta time into account.
    move_and_slide()
```

C#

```
using Godot;

public partial class MyCharacterBody2D : CharacterBody2D
{
    private const float Gravity = 200.0f;
    private const int WalkSpeed = 200;
    public override void _PhysicsProcess(double delta)
    {
        var velocity = Velocity;

        velocity.Y += (float)delta * Gravity;

        if (Input.IsActionPressed("ui_left"))
        {
            velocity.X = -WalkSpeed;
        }
        else if (Input.IsActionPressed("ui_right"))
        {
            velocity.X = WalkSpeed;
        }
        else
        {
            velocity.X = 0;
        }

        Velocity = velocity;

        // "MoveAndSlide" already takes delta time into
account.
        MoveAndSlide();
    }
}
```

And give it a try.

This is a good starting point for a platformer. A more complete demo can be found in the demo zip distributed with the engine, or in the https://github.com/godotengine/godot-demo-projects/tree/master/2d/kinematic_character.

Using SoftBody

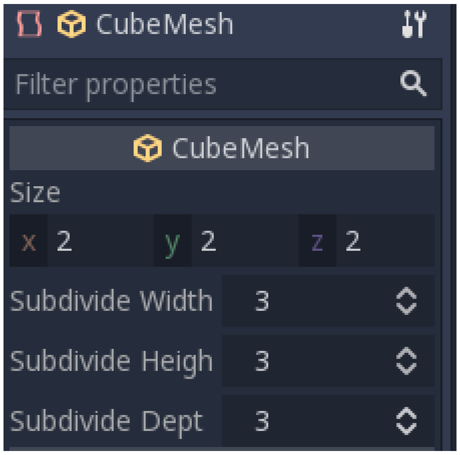
Soft bodies (or *soft-body dynamics*) simulate movement, changing shape and other physical properties of deformable objects. This can for example be used to simulate cloth or to create more realistic characters.

Basic set-up

A SoftBody3D node is used for soft body simulations.

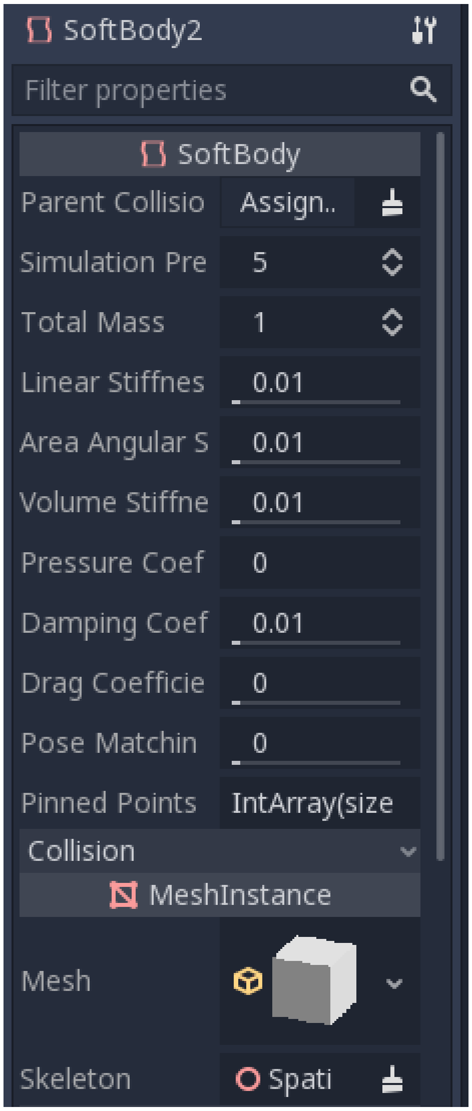
We will create a bouncy cube to demonstrate the setup of a soft body.

Create a new scene with a `Node3D` node as root. Then, create a `Softbody` node. Add a `CubeMesh` in the `mesh` property of the node in the inspector and increase the subdivision of the mesh for simulation.



Size
x 2 y 2 z 2
Subdivide Width 3
Subdivide Heigh 3
Subdivide Dept 3

Set the parameters to obtain the type of soft body you aim for. Try to keep the `Simulation Precision` above 5, otherwise, the soft body may collapse.


SoftBody2

Filter properties

SoftBody

Parent Collisio Assign..

Simulation Pre 5

Total Mass 1

Linear Stiffnes 0.01

Area Angular S 0.01

Volume Stiffne 0.01

Pressure Coef 0

Damping Coef 0.01

Drag Coefficie 0

Pose Matchin 0

Pinned Points IntArray(size

Collision

MeshInstance

Mesh

Skeleton Spati

Note

Handle some parameters with care, as some value can lead to strange results. For example, if the shape is not completely closed and you set pressure to more than 0, the softbody will fly around like a plastic bag under strong wind.

Play the scene to view the simulation.

Tip

To improve the simulation's result, increase the `Simulation Precision`, this will give significant improvement at the cost of performance.

Cloak simulation

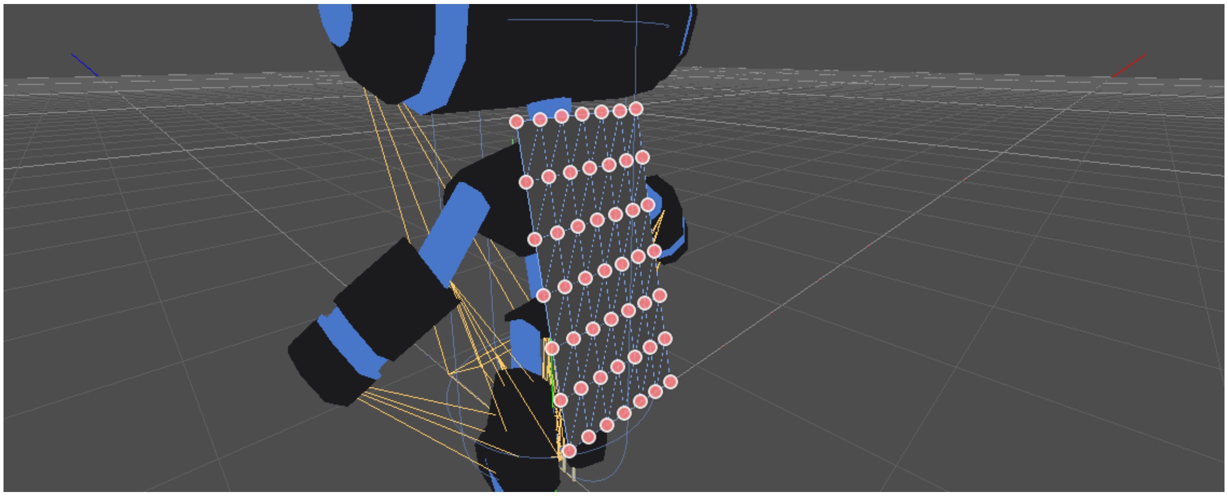
Let's make a cloak in the Platformer3D demo.

Note

You can download the Platformer3D demo on GitHub [https://github.com/godotengine/godot-demo-projects/tree/master/3d/platformer] or the Asset Library [https://godotengine.org/asset-library/asset/2748].

Open the `Player` scene, add a `SoftBody` node and assign a `PlaneMesh` to it.

Open the `PlaneMesh` properties and set the size(x: 0.5 y: 1) then set `Subdivide Width` and `Subdivide Depth` to 5. Adjust the `SoftBody`'s position. You should end up with something like this:



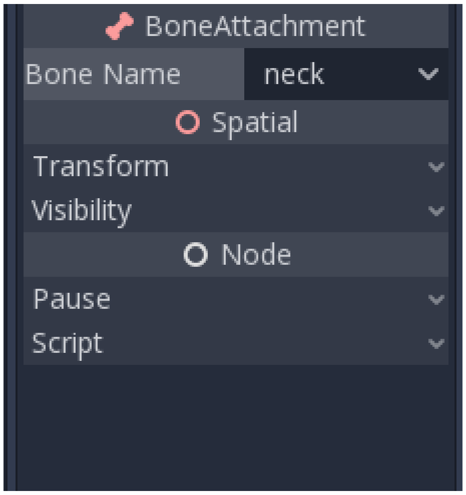
Tip

Subdivision generates a more tessellated mesh for better simulations.

Add a BoneAttachment3D node under the skeleton node and select the Neck bone to attach the cloak to the character skeleton.

Note

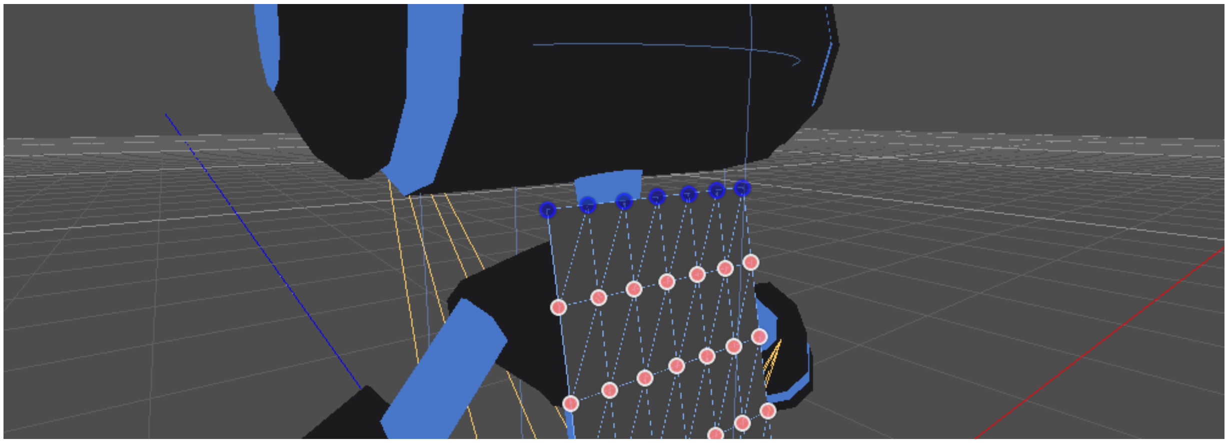
`BoneAttachment3D` node is to attach objects to a bone of an armature. The attached object will follow the bone's movement, weapon of a character can be attached this way.



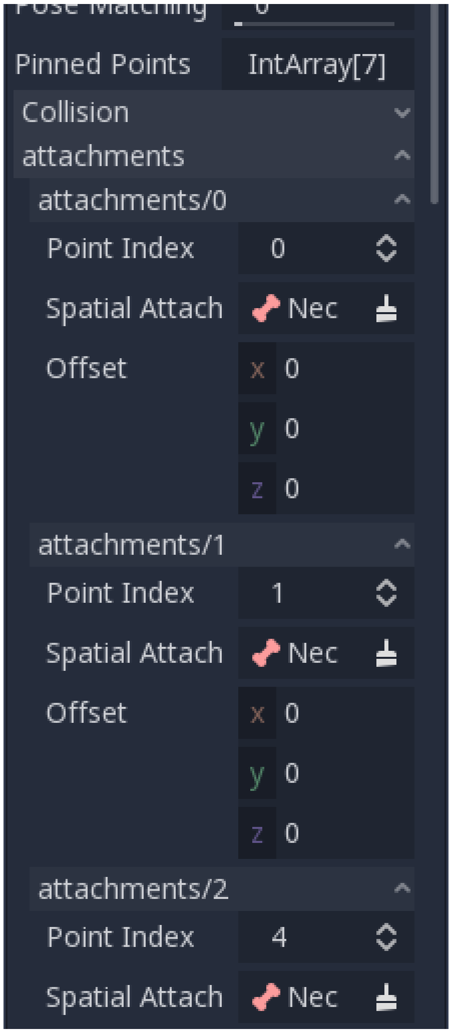
BoneAttachment

Bone Name: neck

To create pinned joints, select the upper vertices in the `SoftBody` node:



The pinned joints can be found in `SoftBody`'s `Attachments` property, choose the `BoneAttachment` as the `SpatialAttachment` for each pinned joints, the pinned joints are now attached to the neck.



Pinned Points IntArray[7]

Collision

attachments

attachments/0

Point Index 0

Spatial Attach Nec

Offset x 0 y 0 z 0

attachments/1

Point Index 1

Spatial Attach Nec

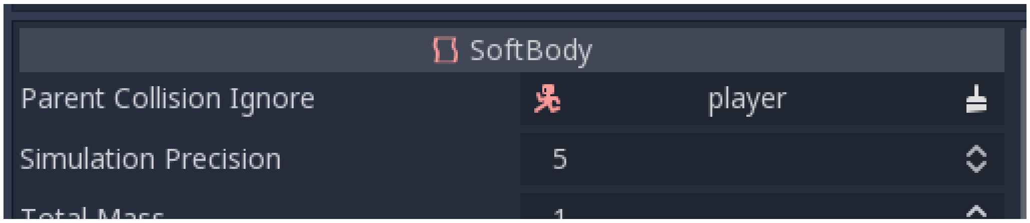
Offset x 0 y 0 z 0

attachments/2

Point Index 4

Spatial Attach Nec

Last step is to avoid clipping by adding the Kinematic Body *Player* to `Parent Collision Ignore` of the `SoftBody`.



SoftBody

Parent Collision Ignore: player

Simulation Precision: 5

Play the scene and the cloak should simulate correctly.

This covers the basic settings of softbody, experiment with the parameters to achieve the effect you are aiming for when making your game.

Collision shapes (2D)

This guide explains:

- The types of collision shapes available in 2D in Godot.
- Using an image converted to a polygon as a collision shape.
- Performance considerations regarding 2D collisions.

Godot provides many kinds of collision shapes, with different performance and accuracy tradeoffs.

You can define the shape of a PhysicsBody2D by adding one or more CollisionShape2Ds or CollisionPolygon2Ds as child nodes. Note that you must add a Shape2D *resource* to collision shape nodes in the Inspector dock.

Note

When you add multiple collision shapes to a single PhysicsBody2D, you don't have to worry about them overlapping. They won't "collide" with each other.

Primitive collision shapes

Godot provides the following primitive collision shape types:

- RectangleShape2D
- CircleShape2D
- CapsuleShape2D
- SegmentShape2D
- SeparationRayShape2D (designed for characters)
- WorldBoundaryShape2D (infinite plane)

You can represent the collision of most smaller objects using one or more primitive shapes. However, for more complex objects, such as a large ship or a whole level, you may need convex or concave shapes instead. More on that below.

We recommend favoring primitive shapes for dynamic objects such as RigidBodies and CharacterBodies as their behavior is the most reliable. They often provide better performance as well.

Convex collision shapes

Warning

Godot currently doesn't offer a built-in way to create 2D convex collision shapes. This section is mainly here for reference purposes.

Convex collision shapes are a compromise between primitive collision shapes and concave collision shapes. They can represent shapes of any complexity, but with an important caveat. As their name implies, an individual shape can only represent a *convex* shape. For instance, a pyramid is *convex*, but a hollow box is *concave*. To define a concave object with a single collision shape, you need to use a concave collision shape.

Depending on the object's complexity, you may get better performance by using multiple convex shapes instead of a concave collision shape. Godot lets you use *convex decomposition* to generate convex shapes that roughly match a hollow object. Note this performance advantage no longer applies after a certain amount of convex shapes. For large and complex objects such as a whole level, we recommend using concave shapes instead.

Concave or trimesh collision shapes

Concave collision shapes, also called trimesh collision shapes, can take any form, from a few triangles to thousands of triangles. Concave shapes are the slowest option but are also the most accurate in Godot. **You can only use concave shapes within StaticBodies.** They will not work with CharacterBodies or RigidBodies unless the RigidBody's mode is Static.

Note

Even though concave shapes offer the most accurate *collision*, contact reporting can be less precise than primitive shapes.

When not using TileMaps for level design, concave shapes are the best approach for a level's collision.

You can configure the CollisionPolygon2D node's *build mode* in the inspector. If it is set to **Solids** (the default), collisions will include the polygon and its contained area. If it is set to **Segments**, collisions will only include the polygon edges.

You can generate a concave collision shape from the editor by selecting a Sprite2D and using the **Sprite2D** menu at the top of the 2D viewport. The Sprite2D menu dropdown exposes an option called **Create CollisionPolygon2D Sibling**. Once you click it, it displays a menu with 3 settings:

- **Simplification:** Higher values will result in a less detailed shape, which improves performance at the cost of accuracy.
- **Shrink (Pixels):** Higher values will shrink the generated collision polygon relative to the sprite's edges.
- **Grow (Pixels):** Higher values will grow the generated collision polygon relative to the sprite's edges. Note that setting Grow and Shrink to equal values may yield different results than leaving both of them on 0.

Note

If you have an image with many small details, it's recommended to create a simplified version and use it to generate the collision polygon. This can result in better performance and game feel, since the player won't be blocked by small, decorative details.

To use a separate image for collision polygon generation, create another Sprite2D, generate a collision polygon sibling from it then remove the Sprite2D node. This way, you can exclude small details from the generated collision.

Performance caveats

You aren't limited to a single collision shape per PhysicsBody. Still, we recommend keeping the number of shapes as low as possible to improve performance, especially for dynamic objects like RigidBodies and CharacterBodies. On top of that, avoid translating, rotating, or scaling CollisionShapes to benefit from the physics engine's internal optimizations.

When using a single non-transformed collision shape in a StaticBody, the engine's *broad phase* algorithm can discard inactive PhysicsBodies. The *narrow phase* will then only have to take into account the active bodies' shapes. If a StaticBody has many collision shapes, the broad phase will fail. The narrow phase, which is slower, must then perform a collision check against each shape.

If you run into performance issues, you may have to make tradeoffs in terms of accuracy. Most games out there don't have a 100% accurate collision. They find creative ways to hide it or otherwise make it unnoticeable during normal gameplay.

Collision shapes (3D)

This guide explains:

- The types of collision shapes available in 3D in Godot.
- Using a convex or a concave mesh as a collision shape.
- Performance considerations regarding 3D collisions.

Godot provides many kinds of collision shapes, with different performance and accuracy tradeoffs.

You can define the shape of a PhysicsBody3D by adding one or more CollisionShape3Ds as child nodes. Note that you must add a Shape3D *resource* to collision shape nodes in the Inspector dock.

Note

When you add multiple collision shapes to a single PhysicsBody, you don't have to worry about them overlapping. They won't "collide" with each other.

Primitive collision shapes

Godot provides the following primitive collision shape types:

- BoxShape3D
- SphereShape3D
- CapsuleShape3D
- CylinderShape3D

You can represent the collision of most smaller objects using one or more primitive shapes. However, for more complex objects, such as a large ship or a whole level, you may need convex or concave shapes instead. More on that below.

We recommend favoring primitive shapes for dynamic objects such as RigidBodies and CharacterBodies as their behavior is the most reliable. They often provide better performance as well.

Convex collision shapes

Convex collision shapes are a compromise between primitive collision shapes and concave collision shapes. They can represent shapes of any complexity, but with an important caveat. As their name implies, an individual shape can only represent a *convex* shape. For instance, a pyramid is *convex*, but a hollow box is *concave*. To define a concave object with a single collision shape, you need to use a concave collision shape.

Depending on the object's complexity, you may get better performance by using multiple convex shapes instead of a concave collision shape. Godot lets you use *convex decomposition* to generate convex shapes that roughly match a hollow object. Note this performance advantage no longer applies after a certain amount of convex shapes. For large and complex objects such as a whole level, we recommend using concave shapes instead.

You can generate one or several convex collision shapes from the editor by selecting a MeshInstance3D and using the **Mesh** menu at the top of the 3D viewport. The editor exposes two generation modes:

- **Create Single Convex Collision Sibling** uses the Quickhull algorithm. It creates one CollisionShape node with an automatically generated convex collision shape. Since it only generates a single shape, it provides good performance and is ideal for small objects.

- **Create Multiple Convex Collision Siblings** uses the V-HACD algorithm. It creates several CollisionShape nodes, each with a convex shape. Since it generates multiple shapes, it is more accurate for concave objects at the cost of performance. For objects with medium complexity, it will likely be faster than using a single concave collision shape.

Concave or trimesh collision shapes

Concave collision shapes, also called trimesh collision shapes, can take any form, from a few triangles to thousands of triangles. Concave shapes are the slowest option but are also the most accurate in Godot. **You can only use concave shapes within StaticBodies.** They will not work with CharacterBodies or RigidBodies unless the RigidBody's mode is Static.

Note

Even though concave shapes offer the most accurate *collision*, contact reporting can be less precise than primitive shapes.

When not using GridMaps for level design, concave shapes are the best approach for a level's collision. That said, if your level has small details, you may want to exclude those from collision for performance and game feel. To do so, you can build a simplified collision mesh in a 3D modeler and have Godot generate a collision shape for it automatically. More on that below

Note that unlike primitive and convex shapes, a concave collision shape doesn't have an actual "volume". You can place objects both *outside* of the shape as well as *inside*.

You can generate a concave collision shape from the editor by selecting a MeshInstance3D and using the **Mesh** menu at the top of the 3D viewport. The editor exposes two options:

- **Create Trimesh Static Body** is a convenient option. It creates a StaticBody containing a concave shape matching the mesh's geometry.
- **Create Trimesh Collision Sibling** creates a CollisionShape node with a concave shape matching the mesh's geometry.

See also

See Importing 3D scenes for information on how to export models for Godot and automatically generate collision shapes on import.

Performance caveats

You aren't limited to a single collision shape per PhysicsBody. Still, we recommend keeping the number of shapes as low as possible to improve performance, especially for dynamic objects like RigidBodies and CharacterBodies. On top of that, avoid translating, rotating, or scaling CollisionShapes to benefit from the physics engine's internal optimizations.

When using a single non-transformed collision shape in a StaticBody, the engine's *broad phase* algorithm can discard inactive PhysicsBodies. The *narrow phase* will then only have to take into account the active bodies' shapes. If a StaticBody has many collision shapes, the broad phase will fail. The narrow phase, which is slower, must then perform a collision check against each shape.

If you run into performance issues, you may have to make tradeoffs in terms of accuracy. Most games out there don't have a 100% accurate collision. They find creative ways to hide it or otherwise make it unnoticeable during normal gameplay.

Large world coordinates

Note

Large world coordinates are mainly useful in 3D projects; they are rarely required in 2D projects. Also, unlike 3D rendering, 2D rendering currently doesn't benefit from increased precision when large world coordinates are enabled.

Why use large world coordinates?

In Godot, physics simulation and rendering both rely on *floating-point* numbers. However, in computing, floating-point numbers have **limited precision and range**. This can be a problem for games with huge worlds, such as space or planetary-scale simulation games.

Precision is the greatest when the value is close to `0.0`. Precision becomes gradually lower as the value increases or decreases away from `0.0`. This occurs every time the floating-point number's *exponent* increases, which happens when the floating-point number surpasses a power of 2 value (2, 4, 8, 16, ...). Every time this occurs, the number's minimum step will *increase*, resulting in a loss of precision.

In practice, this means that as the player moves away from the world origin (`Vector2(0, 0)` in 2D games or `Vector3(0, 0, 0)` in 3D games), precision will decrease.

This loss of precision can result in objects appearing to "vibrate" when far away from the world origin, as the model's position will snap to the nearest value that can be represented in a floating-point number. This can also result in physics glitches that only occur when the player is far from the world origin.

The range determines the minimum and maximum values that can be stored in the number. If the player tries to move past this range, they will simply not be able to. However, in practice, floating-point precision almost always becomes a problem before the range does.

The range and precision (minimum step between two exponent intervals) are determined by the floating-point number type. The *theoretical* range allows extremely high values to be stored in single-precision floats, but with very low precision. In practice, a floating-point type that cannot represent all integer values is not very useful. At extreme values, precision becomes so low that the number cannot even distinguish two separate *integer* values from each other.

This is the range where individual integer values can be represented in a floating-point number:

- **Single-precision float range (represent all integers):** Between -16,777,216 and 16,777,216
- **Double-precision float range (represent all integers):** Between -9 quadrillion and 9 quadrillion

Range	Single step	Double step	Comment
[1; 2]	~0.0000001	~1e-15	Precision becomes greater near 0.0 (this table is abbreviated).
[2; 4]	~0.0000002	~1e-15	

[4; 8]	~0.0000005	~1e-15	
[8; 16]	~0.000001	~1e-14	
[16; 32]	~0.000002	~1e-14	
[32; 64]	~0.000004	~1e-14	
[64; 128]	~0.000008	~1e-13	
[128; 256]	~0.000015	~1e-13	
[256; 512]	~0.00003	~1e-13	
[512; 1024]	~0.00006	~1e-12	
[1024; 2048]	~0.0001	~1e-12	
[2048; 4096]	~0.0002	~1e-12	Maximum *recommended* single-precision range for a first-person 3D game without rendering

			artifacts or physics glitches.
[4096; 8192]	~0.0005	~1e-12	Maximum *recommended* single-precision range for a third-person 3D game without rendering artifacts or physics glitches.
[8192; 16384]	~0.001	~1e-12	
[16384; 32768]	~0.0019	~1e-11	Maximum *recommended* single-precision range for a top-down 3D game without rendering artifacts or physics glitches.
[32768; 65536]	~0.0039	~1e-11	Maximum *recommended* single-precision range for any 3D game. Double precision (large world coordinates) is usually required past this point.
[65536; 131072]	~0.0078	~1e-11	

[131072; 262144]	~0.0156	~1e-10	
> 262144	> ~0.0313	~1e-10 (0.0000000001)	Double-precision remains far more precise than single-precision past this value.

When using single-precision floats, it is possible to go past the suggested ranges, but more visible artifacting will occur and physics glitches will be more common (such as the player not walking straight in certain directions).

See also

See the Demystifying Floating Point Precision [https://blog.demofox.org/2017/11/21/] article for more information.

How large world coordinates work

Large world coordinates (also known as **double-precision physics**) increase the precision level of all floating-point computations within the engine.

By default, float is 64-bit in GDScript, but Vector2, Vector3 and Vector4 are 32-bit. This means that the precision of vector types is much more limited. To resolve this, we can increase the number of bits used to represent a floating-point number in a Vector type. This results in an *exponential* increase in precision, which means the final value is not just twice as precise, but potentially thousands of times more precise at high values. The maximum value that can be represented is also greatly increased by going from a single-precision float to a double-precision float.

To avoid model snapping issues when far away from the world origin, Godot's 3D rendering engine will increase its precision for rendering operations when large world coordinates are enabled. The shaders do not use double-precision floats for performance reasons, but an alternative solution [https://github.com/godotengine/godot/pull/66178] is used to emulate double precision for rendering using single-precision floats.

Note

Enabling large world coordinates comes with a performance and memory usage penalty, especially on 32-bit CPUs. Only enable large world coordinates if you actually need them.

This feature is tailored towards mid-range/high-end desktop platforms. Large world coordinates may not perform well on low-end mobile devices, unless you take steps to reduce CPU usage with other means (such as decreasing the number of physics ticks per second).

On low-end platforms, an *origin shifting* approach can be used instead to allow for large worlds without using double-precision physics and rendering. Origin shifting works with single-precision floats, but it introduces more complexity to game logic, especially in multiplayer games. Therefore, origin shifting is not detailed on this page.

Who are large world coordinates for?

Large world coordinates are typically required for 3D space or planetary-scale simulation games. This extends to games that require supporting *very* fast movement speeds, but also very slow *and* precise movements at times.

On the other hand, it's important to only use large world coordinates when actually required (for performance reasons). Large world coordinates are usually **not** required for:

- 2D games, as precision issues are usually less noticeable.
- Games with small-scale or medium-scale worlds.
- Games with large worlds, but split into different levels with loading sequences in between. You can center each level portion around the world origin to avoid precision issues without a performance penalty.
- Open world games with a *playable on-foot area* not exceeding 8192×8192 meters (centered around the world origin). As shown in the above table, the level of precision remains acceptable within that range, even for a first-person game.

If in doubt, you probably don't need to use large world coordinates in your project. For reference, most modern AAA open world titles don't use a large world coordinates system and still rely on single-precision floats for both rendering and physics.

Enabling large world coordinates

This process requires recompiling the editor and all export template binaries you intend to use. If you only intend to export your project in release mode, you can skip the compilation of debug export templates. In any case, you'll need to compile an editor build so you can test your large precision world without having to export the project every time.

See the Compiling section for compiling instructions for each target platform. You will need to add the `precision=double` SCons option when compiling the editor and export templates.

The resulting binaries will be named with a `.double` suffix to distinguish them from single-precision binaries (which lack any precision suffix). You can then specify the binaries as custom export templates in your project's export presets in the Export dialog.

Compatibility between single-precision and double-precision builds

When saving a *binary* resource using the ResourceSaver singleton, a special flag is stored in the file if the resource was saved using a build that uses double-precision numbers. As a result, all binary resources will change on disk when you switch to a double-precision build and save over them.

Both single-precision and double-precision builds support using the ResourceLoader singleton on resources that use this special flag. This means single-precision builds can load resources saved using double-precision builds and vice versa. Text-based resources don't store a double-precision flag, as they don't require such a flag for correct reading.

Known incompatibilities

- In a networked multiplayer game, the server and all clients should be using the same build type to ensure precision remains consistent across clients. Using different build types *may* work, but various issues can occur.

- The GDExtension API changes in an incompatible way in double-precision builds. This means extensions **must** be rebuilt to work with double-precision builds. On the extension developer's end, the `REAL_T_IS_DOUBLE` define is enabled when building a GDExtension with `precision=double`. `real_t` can be used as an alias for `float` in single-precision builds, and `double` in double-precision builds.

Limitations

Since 3D rendering shaders don't actually use double-precision floats, there are some limitations when it comes to 3D rendering precision:

- Shaders using the `skip_vertex_transform` or `world_vertex_coords` don't benefit from increased precision.
- Triplanar mapping doesn't benefit from increased precision. Materials using triplanar mapping will exhibit visible jittering when far away from the world origin.

2D rendering currently doesn't benefit from increased precision when large world coordinates are enabled. This can cause visible model snapping to occur when far away from the world origin (starting from a few million pixels at typical zoom levels). 2D physics calculations will still benefit from increased precision though.

Troubleshooting physics issues

When working with a physics engine, you may encounter unexpected results.

While many of these issues can be resolved through configuration, some of them are the result of engine bugs. For known issues related to the physics engine, see open physics-related issues on GitHub [https://github.com/godotengine/godot/issues?q=is%3Aopen+is%3Aissue+label%3Atopic%3Aphysics]. Looking through closed issues [https://github.com/godotengine/godot/issues?q=+is%3Aclosed+is%3Aissue+label%3Atopic%3Aphysics] can also help answer questions related to physics engine behavior.

Objects are passing through each other at high speeds

This is known as *tunneling*. Enabling **Continuous CD** in the RigidBody properties can sometimes resolve this issue. If this does not help, there are other solutions you can try:

- Make your static collision shapes thicker. For example, if you have a thin floor that the player can't get below in some way, you can make the collider thicker than the floor's visual representation.
- Modify your fast-moving object's collision shape depending on its movement speed. The faster the object moves, the larger the collision shape should extend outside of the object to ensure it can collide with thin walls more reliably.

- Increase **Physics Ticks Per Second** in the advanced Project Settings. While this has other benefits (such as more stable simulation and reduced input lag), this increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Stacked objects are unstable and wobbly

Despite seeming like a simple problem, stable RigidBody simulation with stacked objects is difficult to implement in a physics engine. This is caused by integrating forces going against each other. The more stacked objects are present, the stronger the forces will be against each other. This eventually causes the simulation to become wobbly, making the objects unable to rest on top of each other without moving.

Increasing the physics simulation rate can help alleviate this issue. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Scaled physics bodies or collision shapes do not collide correctly

Godot does not currently support scaling of physics bodies or collision shapes. As a workaround, change the collision shape's extents instead of changing its scale. If you want the visual representation's scale to change as well, change the scale of the underlying visual representation (Sprite2D, MeshInstance3D, …) and change the collision shape's extents separately. Make sure the collision shape is not a child of the visual representation in this case.

Since resources are shared by default, you'll have to make the collision shape resource unique if you don't want the change to be applied to all nodes using the same collision shape resource in the scene. This can be done by calling `duplicate()` in a script on the collision shape resource *before* changing its size.

Thin objects are wobbly when resting on the floor

This can be due to one of two causes:

- The floor's collision shape is too thin.
- The RigidBody's collision shape is too thin.

In the first case, this can be alleviated by making the floor's collision shape thicker. For example, if you have a thin floor that the player can't get below in some way, you can make the collider thicker than the floor's visual representation.

In the second case, this can usually only be resolved by increasing the physics simulation rate (as making the shape thicker would cause a disconnect between the RigidBody's visual representation and its collision).

In both cases, increasing the physics simulation rate can also help alleviate this issue. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that this increases CPU utilization and may not be viable for mobile/web platforms. Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Cylinder collision shapes are unstable

During the transition from Bullet to GodotPhysics in Godot 4, cylinder collision shapes had to be reimplemented from scratch. However, cylinder collision shapes are one of the most difficult shapes to support, which is why many other physics engines don't provide any support for them. There are several known bugs with cylinder collision shapes currently.

We recommend using box or capsule collision shapes for characters for now. Boxes generally provide the best reliability, but have the downside of making the character take more space diagonally. Capsule collision shapes do not have this downside, but their shape can make precision platforming more difficult.

VehicleBody simulation is unstable, especially at high speeds

When a physics body moves at a high speed, it travels a large distance between each physics step. For instance, when using the 1 unit = 1 meter convention in 3D, a vehicle moving at 360 km/h will travel 100 units per second. With the default physics simulation rate of 60 Hz, the vehicle moves by ~1.67 units each physics tick. This means that small objects may be ignored entirely by the vehicle (due to tunneling), but also that the simulation has little data to work with in general at such a high speed.

Fast-moving vehicles can benefit a lot from an increased physics simulation rate. To do so, increase **Physics Ticks Per Second** in the advanced Project Settings. Note that this increases CPU utilization and may not be viable for mobile/web platforms.

Multipliers of the default value of `60` (such as `120`, `180` or `240`) should be preferred for a smooth appearance on most displays.

Collision results in bumps when an object moves across tiles

This is a known issue in the physics engine caused by the object bumping on a shape's edges, even though that edge is covered by another shape. This can occur in both 2D and 3D.

The best way to work around this issue is to create a "composite" collider. This means that instead of individual tiles having their collision, you create a single collision shape representing the collision for a group of tiles. Typically, you should split composite colliders on a per-island basis (which means each group of touching tiles gets its own collider).

Using a composite collider can also improve physics simulation performance in certain cases. However, since the composite collision shape is much more complex, this may not be a net performance win in all cases.

Framerate drops when an object touches another object

This is likely due to one of the objects using a collision shape that is too complex. Convex collision shapes should use a number of shapes as low as possible for performance reasons. When relying on Godot's automatic generation, it's possible that you ended up with dozens if not hundreds of shapes created for a single convex shape collision resource.

In some cases, replacing a convex collider with a couple of primitive collision shapes (box, sphere, or capsule) can deliver better performance.

This issue can also occur with StaticBodies that use very detailed trimesh (concave) collisions. In this case, use a simplified representation of the level geometry as a collider. Not only this will improve physics simulation performance significantly, but this can also improve stability by letting you remove small fixtures and crevices from being considered by collision.

Physics simulation is unreliable when far away from the world origin

This is caused by floating-point precision errors, which become more pronounced as the physics simulation occurs further away from the world origin. This issue also affects rendering, which results in wobbly camera movement when far away from the world origin. See Large world coordinates for more information.

Platform-specific

- Android
- iOS plugins
- Web
- Console support in Godot

Android

- Godot Android library
- Godot Android plugins
- Android in-app purchases

Godot Android library

The Godot Engine for Android platforms is designed to be used as an Android library [https://developer.android.com/studio/projects/android-library]. This architecture enables several key features on Android platforms:

- Ability to integrate the Gradle build system within the Godot Editor, which provides the ability to leverage more components from the Android ecosystem such as libraries and tools
- Ability to make the engine portable and embeddable:
  - Key in enabling the port of the Godot Editor to Android and mobile XR devices
  - Key in allowing the integration and reuse of Godot's capabilities within existing codebase

Below we describe some of the use-cases and scenarios this architecture enables.

Using the Godot Android library

The Godot Android library is packaged as an AAR archive file and hosted on MavenCentral [https://central.sonatype.com/artifact/org.godotengine/godot] along with its documentation [https://javadoc.io/doc/org.godotengine/godot/latest/index.html].

It provides access to Godot APIs and capabilities on Android platforms for the following non-exhaustive use-cases.

Godot Android plugins

Android plugins are powerful tools to extend the capabilities of the Godot Engine by tapping into the functionality provided by Android platforms and ecosystem.

An Android plugin is an Android library with a dependency on the Godot Android library which the plugin uses to integrate into the engine's lifecycle and to access Godot APIs, granting it powerful capabilities such as GDExtension support which allows to update / mod the engine behavior as needed.

For more information, see Godot Android plugins.

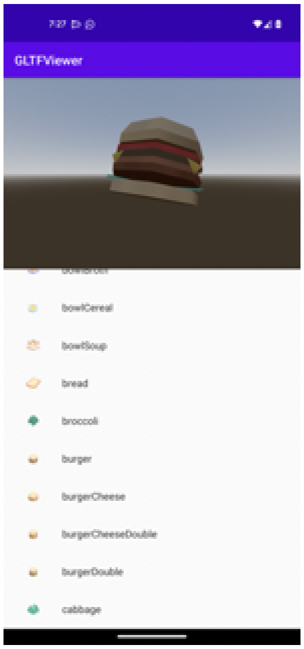
Embedding Godot in existing Android projects

The Godot Engine can be embedded within existing Android applications or libraries, allowing developers to leverage mature and battle-tested code and libraries better suited to a specific task.

The hosting component is responsible for driving the engine lifecycle via Godot's Android APIs. These APIs can also be used to provide bidirectional communication between the host and the embedded Godot instance allowing for greater control over the desired experience.

We showcase how this is done using a sample Android app that embeds the Godot Engine as an Android view, and uses it to render 3D glTF models.

The GLTF Viewer [https://github.com/m4gr3d/Godot-Android-Samples/tree/master/apps/gltf_viewer] sample app uses an Android RecyclerView component [https://developer.android.com/develop/ui/views/layout/recyclerview] to create a list of glTF items, populated from Kenney's Food Kit pack [https://kenney.nl/assets/food-kit]. When an item on the list is selected, the app's logic interacts with the embedded Godot Engine to render the selected glTF item as a 3D model.


7:27
GLTFViewer
bowlCereal
bowlSoup
bread
broccoli
burger
burgerCheese
burgerCheeseDouble
burgerDouble
cabbage

The sample app source code can be found on GitHub [https://github.com/m4gr3d/Godot-Android-Samples/tree/master/apps/gltf_viewer]. Follow the instructions on its README [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/README.md] to build and install it.

Below we break-down the steps used to create the GLTF Viewer app.

Warning

Currently only a single instance of the Godot Engine is supported per process. You can configure the process the Android Activity runs under using the android:process attribute [https://developer.android.com/guide/topics/manifest/activity-element#proc].

Warning

Automatic resizing / orientation configuration events are not supported and may cause a crash. You can disable those events:

- By locking to a specific orientation using the android:screenOrientation attribute [https://developer.android.com/guide/topics/manifest/activity-element#screen].
- By declaring that the Activity will handle these configuration events using the android:configChanges attribute [https://developer.android.com/guide/topics/manifest/activity-element#config].

1. Create the Android app

Note

The Android sample app was created using Android Studio [https://developer.android.com/studio] and using Gradle [https://developer.android.com/build] as the build system.

The Android ecosystem provides multiple tools, IDEs, build systems for creating Android apps so feel free to use what you're familiar with, and update the steps below accordingly (contributions to this documentation are welcomed as well!).

- Set up an Android application project. It may be a brand new empty project, or an existing project
- Add the maven dependency for the Godot Android library [https://central.sonatype.com/artifact/org.godotengine/godot]
  - If using `gradle`, add the following to the `dependency` section of the app's gradle build file. Make sure to update

`<version>` to the latest version of the Godot Android library:

```
implementation("org.godotengine:godot:<version>")
```

- If using `gradle`, include the following `aaptOptions` configuration under the `android > defaultConfig` section of the app's gradle build file. Doing so allows `gradle` to include Godot's hidden directories when building the app binary.
    - If your build system does not support including hidden directories, you can [configure the Godot project to not use hidden directories](https://docs.godotengine.org/en/stable/classes/class_projectsettings.html#class-projectsettings-property-application-config-use-hidden-project-data-directory) [https://docs.godotengine.org/en/stable/classes/class_projectsettings.html#class-projectsettings-property-application-config-use-hidden-project-data-directory] by deselecting `Project Settings... > Application > Config > Use Hidden Project Data Directory`.

```
android {

  defaultConfig {
      // The default ignore pattern for the 'assets' directory includes hidden files and
      // directories which are used by Godot projects, so we override it with the following.
      aaptOptions {
          ignoreAssetsPattern "!.svn:!.git:!.gitignore:!.ds_store:!*.scc:<dir>_*:!CVS:!thumbs.db:!picasa.ini:!*~"
      }
    ...
```

- Create / update the application's Activity that will be hosting the Godot Engine instance. For the sample app, this is [MainActivity](https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/src/main/java/fhuyakou/godot/app/android/gltfviewer/MainActivity.kt) [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/src/main/java/fhuyakou/godot/app/android/gltfviewer/MainActivity.kt]

- The host Activity should implement the GodotHost interface [https://github.com/godotengine/godot/blob/master/platform/android/java/lib/src/org/godotengine/godot/GodotHost.java]
- The sample app uses Fragments [https://developer.android.com/guide/fragments] to organize its UI, so it uses GodotFragment [https://github.com/godotengine/godot/blob/master/platform/android/java/lib/src/org/godotengine/godot/GodotFragment.java], a fragment component provided by the Godot Android library to automatically host and manage the Godot Engine instance.

```
private var godotFragment: GodotFragment? = null

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_main)

    val currentGodotFragment =
supportFragmentManager.findFragmentById(R.id.godot_fragment
_container)
    if (currentGodotFragment is GodotFragment) {
        godotFragment = currentGodotFragment
    } else {
        godotFragment = GodotFragment()
        supportFragmentManager.beginTransaction()
            .replace(R.id.godot_fragment_container,
godotFragment!!)
            .commitNowAllowingStateLoss()
    }

    ...
```

Note

The Godot Android library also provide GodotActivity [https://github.com/godotengine/godot/blob/master/platform/android/java/lib/src/org/godotengine/godot/GodotActivity.kt], an Activity component that can be extended to automatically host and manage the Godot Engine instance.

Alternatively, applications can directly create a Godot [https://github.com/godotengine/godot/blob/master/platform/android/java/lib/src/org/godotengine/godot/Godot.kt] instance, host and manage it themselves.

- Using GodotHost#getHostPlugins(...) [https://github.com/m4gr3d/Godot-Android-Samples/blob/0e3440f357f8be5b4c63a4fe75766793199a99d0/apps/gltf_viewer/src/main/java/fhuyakou/godot/app/android/gltfviewer/MainActivity.kt#L55], the sample app creates a runtime GodotPlugin instance [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/src/main/java/fhuyakou/godot/app/android/gltfviewer/AppPlugin.kt] that's used to send signals to the `gdscript` logic
    - The runtime `GodotPlugin` can also be used by `gdscript` logic to access JVM methods. For more information, see Godot Android plugins.
- Add any additional logic that will be used by your application
    - For the sample app, this includes adding the ItemsSelectionFragment fragment [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/src/main/java/fhuyakou/godot/app/android/gltfviewer/ItemsSelectionFragment.kt] (and related classes), a fragment used to build and show the list of glTF items
- Open the `AndroidManifest.xml` file, and configure the orientation if needed using the android:screenOrientation attribute [https://developer.android.com/guide/topics/manifest/activity-element#screen]
    - If needed, disable automatic resizing / orientation configuration changes using the android:configChanges attribute

[https://developer.android.com/guide/topics/manifest/activity-element#config]

```
<activity android:name=".MainActivity"
    android:screenOrientation="fullUser"

android:configChanges="orientation|screenSize|smallestScreenSiz
e|screenLayout"
    android:exported="true">

    ...
</activity>
```

2. Create the Godot project

Note

On Android, Godot's project files are exported to the `assets` directory of the generated `apk` binary.

We leverage that architecture to bind our Android app and Godot project together by creating the Godot project in the Android app's `assets` directory.

Note that it's also possible to create the Godot project in a separate directory and export it as a PCK or ZIP file [https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html#pck-versus-zip-pack-file-formats] to the Android app's `assets` directory. Using this approach requires passing the `--main-pack <pck_or_zip_filepath_relative_to_assets_dir>` argument to the hosted Godot Engine instance using GodotHost#getCommandLine() [https://github.com/godotengine/godot/blob/6916349697a4339216469e9bf5899b983d78db07/platform/android/java/lib/src/org/godotengine/godot/GodotHost.java#L45].

The instructions below and the sample app follow the first approach of creating the Godot project in the Android app's `assets` directory.

- As mentioned in the **note** above, open the Godot Editor and create a Godot project directly (no subfolder) in the `assets` directory of the Android application project
  - See the sample app's Godot project [https://github.com/m4gr3d/Godot-Android-Samples/tree/master/apps/gltf_viewer/src/main/assets] for reference
- Configure the Godot project as desired
  - Make sure the orientation [https://docs.godotengine.org/en/stable/classes/class_projectsettings.html#class-projectsettings-property-display-window-handheld-orientation] set for the Godot project matches the one set in the Android app's manifest
  - For Android, make sure textures/vram_compression/import_etc2_astc [https://docs.godotengine.org/en/stable/classes/class_projectsettings.html#class-projectsettings-property-rendering-textures-vram-compression-import-etc2-astc] is set to *true*
- Update the Godot project script logic as needed
  - For the sample app, the script logic [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/src/main/assets/main.gd] queries for the runtime `GodotPlugin` instance and uses it to register for signals fired by the app logic
  - The app logic fires a signal every time an item is selected in the list. The signal contains the filepath of the glTF model, which is used by the `gdscript` logic to render the model.

```
extends Node3D

# Reference to the gltf model that's currently being shown.
var current_gltf_node: Node3D = null

func _ready():
  # Default asset to load when the app starts
  _load_gltf("res://gltfs/food_kit/turkey.glb")

  var appPlugin = Engine.get_singleton("AppPlugin")
  if appPlugin:
    print("App plugin is available")

    # Signal fired from the app logic to update the gltf
model being shown
    appPlugin.connect("show_gltf", _load_gltf)
  else:
    print("App plugin is not available")


# Load the gltf model specified by the given path
func _load_gltf(gltf_path: String):
  if current_gltf_node != null:
    remove_child(current_gltf_node)

  current_gltf_node = load(gltf_path).instantiate()

  add_child(current_gltf_node)
```

3. Build and run the app

Once you complete configuration of your Godot project, build and run the Android app. If set up correctly, the host Activity will initialize the embedded Godot Engine on startup. The Godot Engine will check the `assets` directory for project files to load (unless configured to look for a `main pack`), and will proceed to run the project.

While the app is running on device, you can check Android logcat [https://developer.android.com/studio/debug/logcat] to investigate any errors or crashes.

For reference, check the build and install instructions [https://github.com/m4gr3d/Godot-Android-Samples/blob/master/apps/gltf_viewer/README.md] for the GLTF Viewer sample app.

Godot Android plugins

Introduction

Android plugins are powerful tools to extend the capabilities of the Godot engine by tapping into the functionality provided by Android platforms and ecosystem.

For example in Godot 4, Android plugins are used to support multiple Android-based XR platforms without encumbering the core codebase with vendor specific code or binaries.

Android plugin

Version 1 (v1) of the Android plugin system was introduced in Godot 3 and compatible with Godot 4.0 and 4.1. That version allowed developers to augment the Godot engine with Java, Kotlin and native functionality.

Starting in Godot 4.2, Android plugins built on the v1 architecture are now deprecated. Instead, Godot 4.2 introduces a new **Version 2 (v2)** architecture for Android plugins.

v2 Architecture

Note

Godot Android plugin leverages the Gradle build system.

Building on the previous v1 architecture, Android plugins continue to be derived from the Android archive library [https://developer.android.com/studio/projects/android-library#aar-contents].

At its core, a Godot Android plugin v2 is an Android library with a dependency on the Godot Android library, and a custom Android library manifest.

This architecture allows Android plugins to extend the functionality of the engine with:

- Android platform APIs
- Android libraries
- Kotlin and Java libraries
- Native libraries (via JNI)
- GDExtension libraries

Each plugin has an init class extending from the GodotPlugin [https://github.com/godotengine/godot/blob/0a7f75ec7b465604b6496c8f5f1d638aed250d6d/platform/android/java/lib/src/org/godotengine/godot/plugin/GodotPlugin.java#L80] class which is provided by the Godot Android library.

The `GodotPlugin` class provides APIs to access the running Godot instance and hook into its lifecycle. It is loaded at runtime by the Godot engine.

v2 Packaging format

v1 Android plugins required a custom `gdap` configuration file that was used by the Godot Editor to detect and load them. However this approach had several drawbacks, primary ones being that it lacked flexibility and departed from the existing Godot EditorExportPlugin format, delivery and installation flow [https://docs.godotengine.org/en/stable/tutorials/plugins/editor/installing_plugins.html].

This has been resolved for v2 Android plugins by deprecating the `gdap` packaging and configuration mechanism in favor of the existing Godot `EditorExportPlugin` packaging format. The `EditorExportPlugin` API in turn has been extended to properly support Android plugins.

Building a v2 Android plugin

A github project template **is provided** at https://github.com/m4gr3d/Godot-Android-Plugin-Template as a **quickstart for building Godot Android plugins for Godot 4.2+**. You can follow the template README [https://github.com/m4gr3d/Godot-Android-Plugin-Template#readme] to set up your own Godot Android plugin project.

To provide further understanding, here is a break-down of the steps used to create the project template:

1. Create an Android library module using these instructions [https://developer.android.com/studio/projects/android-library]

2. Add the Godot Android library as a dependency by updating the module's `gradle` build file [https://github.com/m4gr3d/Godot-Android-Plugin-Template/blob/main/plugin/build.gradle.kts#L42]:

```
dependencies {
    implementation("org.godotengine:godot:4.2.0.stable")
}
```

The Godot Android library is hosted on MavenCentral [https://central.sonatype.com/artifact/org.godotengine/godot], and updated for each release.

3. Create GodotAndroidPlugin [https://github.com/m4gr3d/Godot-Android-Plugin-Template/blob/a01286b4cb459133bf07b11dfabdfd3980268797/plugin/src/ma

in/java/org/godotengine/plugin/android/template/GodotAndroidPlugin.kt#L10], an init class for the plugin extending GodotPlugin [https://github.com/godotengine/godot/blob/0a7f75ec7b465604b6496c8f5f1d638aed250d6d/platform/android/java/lib/src/org/godotengine/godot/plugin/GodotPlugin.java#L80].

   - If the plugin exposes Kotlin or Java methods to be called from GDScript, they must be annotated with @UsedByGodot [https://github.com/godotengine/godot/blob/0a7f75ec7b465604b6496c8f5f1d638aed250d6d/platform/android/java/lib/src/org/godotengine/godot/plugin/UsedByGodot.java#L45]. The name called from GDScript **must match the method name exactly**. There is **no** coercing `snake_case` to `camelCase`. For example, from GDScript:

     ```
     if Engine.has_singleton("MyPlugin"):
         var singleton =
     Engine.get_singleton("MyPlugin")
         print(singleton.myPluginFunction("World"))
     ```

   - If the plugin uses signals [https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html], the init class must return the set of signals used by overriding GodotPlugin::getPluginSignals() [https://github.com/godotengine/godot/blob/fa3428ff25bc577d2a3433090478a6d615567056/platform/android/java/lib/src/org/godotengine/godot/plugin/GodotPlugin.java#L302]. To emit signals, the plugin can use the GodotPlugin::emitSignal(...) method [https://github.com/godotengine/godot/blob/0a7f75ec7b465604b6496c8f5f1d638aed250d6d/platform/android/java/lib/src/org/godotengine/godot/plugin/GodotPlugin.java#L317].

4. Update the plugin `AndroidManifest.xml` file [https://github.com/m4gr3d/Godot-Android-Plugin-

Template/blob/main/plugin/src/main/AndroidManifest.xml] with the following meta-data:

```
<meta-data
    android:name="org.godotengine.plugin.v2.[PluginName]"
    android:value="[plugin.init.ClassFullName]" />
```

Where:

- `PluginName` is the name of the plugin
- `plugin.init.ClassFullName` is the full component name (package + class name) of the plugin init class (e.g: `org.godotengine.plugin.android.template.GodotAndroidPlugin`).

5. Create the EditorExportPlugin configuration [https://github.com/m4gr3d/Godot-Android-Plugin-Template/tree/main/plugin/export_scripts_template] to package the plugin. The steps used to create the configuration can be seen in the Packaging a v2 Android plugin section.

Building a v2 Android plugin with GDExtension capabilities

Similar to GDNative support in v1 Android plugins, v2 Android plugins support the ability to integrate GDExtension capabilities.

A github project template is provided at https://github.com/m4gr3d/GDExtension-Android-Plugin-Template as a quickstart for building GDExtension Android plugins for Godot 4.2+. You can follow the template's README [https://github.com/m4gr3d/GDExtension-Android-Plugin-Template#readme] to set up your own Godot Android plugin project.

Migrating a v1 Android plugin to v2

Use the following steps if you have a v1 Android plugin you want to migrate to v2:

1. Update the plugin's manifest file:
   - Change the `org.godotengine.plugin.v1` prefix to `org.godotengine.plugin.v2`
2. Update the Godot Android library build dependency:
   - You can continue using the `godot-lib.<version>.<status>.aar` binary from [Godot's download page](https://godotengine.org/download) [https://godotengine.org/download] if that's your preference. Make sure it's updated to the latest stable version.
   - Or you can switch to the MavenCentral provided dependency:

     ```
     dependencies {

     implementation("org.godotengine:godot:4.2.0.stable")
     }
     ```
3. After updating the Godot Android library dependency, sync or build the plugin and resolve any compile errors:
   - The `Godot` instance provided by `GodotPlugin::getGodot()` no longer has access to a `android.content.Context` reference. Use `GodotPlugin::getActivity()` instead.
4. Delete the `gdap` configuration file(s) and follow the instructions in the [Packaging a v2 Android plugin]() section to set up the plugin configuration.

Packaging a v2 Android plugin

As mentioned, a v2 Android plugin is now provided to the Godot Editor as an `EditorExportPlugin` plugin, so it shares a lot of the same packaging steps [https://docs.godotengine.org/en/stable/tutorials/plugins/editor/making_plugins.html#creating-a-plugin].

1. Add the plugin output binaries within the plugin directory (e.g: in `addons/<plugin_name>/`)
2. Add the tool script [https://docs.godotengine.org/en/stable/tutorials/plugins/editor/making_plugins.html#the-script-file] for the export functionality within the plugin directory (e.g: in `addons/<plugin_name>/`)
    - The created script must be a `@tool` script, or else it will not work properly
    - The export tool script is used to configure the Android plugin and hook it within the Godot Editor's export process. It should look something like this:

```
@tool
extends EditorPlugin

# A class member to hold the editor export plugin
during its lifecycle.
var export_plugin : AndroidExportPlugin

func _enter_tree():
    # Initialization of the plugin goes here.
    export_plugin = AndroidExportPlugin.new()
    add_export_plugin(export_plugin)


func _exit_tree():
    # Clean-up of the plugin goes here.
    remove_export_plugin(export_plugin)
    export_plugin = null
```

```
class AndroidExportPlugin extends
EditorExportPlugin:
    # Plugin's name.
    var _plugin_name = "<plugin_name>"

    # Specifies which platform is supported by the
plugin.
    func _supports_platform(platform):
        if platform is EditorExportPlatformAndroid:
            return true
        return false

    # Return the paths of the plugin's AAR binaries
relative to the 'addons' directory.
    func _get_android_libraries(platform, debug):
        if debug:
            return PackedStringArray(["
<paths_to_debug_android_plugin_aar_binaries>"])
        else:
            return PackedStringArray(["
<paths_to_release_android_plugin_aar_binaries>"])

    # Return the plugin's name.
    func _get_name():
        return _plugin_name
```

- Here are the set of EditorExportPlugin APIs [https://docs.godotengine.org/en/stable/classes/class_editorexporttplugin.html] most relevant to use in this tool script:
    - _supports_platform [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-supports-platform]: returns `true` if the plugin supports the given platform. For Android plugins, this must return `true` when `platform` is EditorExportPlatformAndroid

[https://docs.godotengine.org/en/stable/classes/class_editorexportplatformandroid.html]

- _get_android_libraries [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-libraries]: retrieve the local paths of the Android libraries binaries (AAR files) provided by the plugin
- _get_android_dependencies [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-dependencies]: retrieve the set of Android maven dependencies (e.g: *org.godot.example:my-plugin:0.0.0*) provided by the plugin
- _get_android_dependencies_maven_repos [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-dependencies-maven-repos]: retrieve the urls of the maven repos for the android dependencies provided by `_get_android_dependencies`
- _get_android_manifest_activity_element_contents [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-manifest-activity-element-contents]: update the contents of the *<activity>* element in the generated Android manifest
- _get_android_manifest_application_element_contents [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-manifest-application-element-

    contents]: update the contents of the *<application>* element in the generated Android manifest

    - _get_android_manifest_element_contents [https://docs.godotengine.org/en/latest/classes/class_editorexportplugin.html#class-editorexportplugin-method-get-android-manifest-element-contents]: update the contents of the *<manifest>* element in the generated Android manifest

  The `_get_android_manifest_*` methods allow the plugin to automatically provide changes to the app's manifest which are preserved when the Godot Editor is updated, resolving a long standing issue with v1 Android plugins.

3. Create a `plugin.cfg`. This is an INI file with metadata about your plugin:

```
[plugin]

name="<plugin_name>"
description="<plugin_description>"
author="<plugin_author>"
version="<plugin_version>"
script="<relative_path_to_the_export_tool_script>"
```

For reference, here is the folder structure for the Godot Android plugin project template [https://github.com/m4gr3d/Godot-Android-Plugin-Template/tree/main/plugin/export_scripts_template]. At build time, the contents of the `export_scripts_template` directory as well as the generated plugin binaries are copied to the `addons/<plugin_name>` directory:

```
export_scripts_template/
|
+--export_plugin.gd         # export plugin tool script

```

```
|
+--plugin.cfg                  # plugin INI file
```

Packaging a v2 Android plugin with GDExtension capabilities

For GDExtension, we follow the same steps as for Packaging a v2 Android plugin and add the GDExtension config file [https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/gdextension_cpp_example.html#using-the-gdextension-module] in the same location as `plugin.cfg`.

For reference, here is the folder structure for the GDExtension Android plugin project template [https://github.com/m4gr3d/GDExtension-Android-Plugin-Template/tree/main/plugin/export_scripts_template]. At build time, the contents of the `export_scripts_template` directory as well as the generated plugin binaries are copied to the `addons/<plugin_name>` directory:

```
export_scripts_template/
|
+--export_plugin.gd            # export plugin tool script
|
+--plugin.cfg                  # plugin INI file
|
+--plugin.gdextension          # GDExtension config file
```

Here is what the `plugin.gdextension` config file should look like:

```
[configuration]

entry_symbol = "plugin_library_init"
compatibility_minimum = "4.2"
android_aar_plugin = true

[libraries]

android.debug.arm64 =
"res://addons/GDExtensionAndroidPluginTemplate/bin/debug/arm64-
v8a/libGDExtensionAndroidPluginTemplate.so"
android.release.arm64 =
"res://addons/GDExtensionAndroidPluginTemplate/bin/release/arm6
4-v8a/libGDExtensionAndroidPluginTemplate.so"
...
```

Of note is the `android_aar_plugin` field that specifies this GDExtension module is provided as part of a v2 Android plugin. During the export process, this will indicate to the Godot Editor that the GDExtension native shared libraries are exported by the Android plugin AAR binaries.

For GDExtension Android plugins, the plugin init class must override GodotPlugin::getPluginGDExtensionLibrariesPaths() [https://github.com/godotengine/godot/blob/0a7f75ec7b465604b6496c8f5f1d638aed250d6d/platform/android/java/lib/src/org/godotengine/godot/plugin/GodotPlugin.java#L277], and return the paths to the bundled GDExtension libraries config files (`*.gdextension`).

The paths must be relative to the Android library's `assets` directory. At runtime, the plugin will provide these paths to the Godot engine which will use them to load and initialize the bundled GDExtension libraries.

Using a v2 Android plugin

Note

- Godot 4.2 or higher is required
- v2 Android plugin requires the use of the Gradle build process [https://docs.godotengine.org/en/stable/classes/class_editorexportplatformandroid.html#class-editorexportplatformandroid-property-gradle-build-use-gradle-build].

- The provided github project templates include demo Godot projects for quick testing.

1. Copy the plugin's output directory (`addons/<plugin_name>`) to the target Godot project's directory
2. Open the project in the Godot Editor; the Editor should detect the plugin
3. Navigate to `Project` -> `Project Settings...` -> `Plugins`, and ensure the plugin is enabled
4. Install the Godot Android build template by clicking on `Project` -> `Install Android Build Template...`
5. Navigate to `Project` -> `Export...`
6. In the `Export` window, create an `Android export preset`
7. In the `Android export preset`, scroll to `Gradle Build` and set `Use Gradle Build` to `true`
8. Update the project's scripts as needed to access the plugin's functionality. For example:

```
if Engine.has_singleton("MyPlugin"):
        var singleton = Engine.get_singleton("MyPlugin")
        print(singleton.myPluginFunction("World"))
```

9. Connect an Android device to your machine and run the project on it

Using a v2 Android plugin as an Android library

Since they are also Android libraries, Godot v2 Android plugins can be stripped from their `EditorExportPlugin` packaging and provided as raw `AAR` binaries for use as libraries alongside the Godot Android library by Android apps.

If targeting this use-case, make sure to include additional instructions for how the `AAR` binaries should be included (e.g: custom additions to the Android app's manifest).

Reference implementations

- Godot Android Plugins Samples [https://github.com/m4gr3d/Godot-Android-Samples/tree/master/plugins]
- Godot Android Plugin Template [https://github.com/m4gr3d/Godot-Android-Plugin-Template]
- GDExtension Android Plugin Template [https://github.com/m4gr3d/GDExtension-Android-Plugin-Template]
- Godot OpenXR Loaders [https://github.com/GodotVR/godot_openxr_loaders]

Tips and Guidelines

Simplify access to the exposed Java / Kotlin APIs

To make it easier to access the exposed Java / Kotlin APIs in the Godot Editor, it's recommended to provide one (or multiple) gdscript wrapper class(es) for your plugin users to interface with.

For example:

```
class_name PluginInterface extends Object

## Interface used to access the functionality provided by this plugin.
```

```
var _plugin_name = "GDExtensionAndroidPluginTemplate"
var _plugin_singleton

func _init():
    if Engine.has_singleton(_plugin_name):
        _plugin_singleton = Engine.get_singleton(_plugin_name)
    else:
        printerr("Initialization error: unable to access the
java logic")

## Print a 'Hello World' message to the logcat.
func helloWorld():
    if _plugin_singleton:
        _plugin_singleton.helloWorld()
    else:
        printerr("Initialization error")
```

Support using the GDExtension functionality in the Godot Editor

If planning to use the GDExtension functionality in the Godot Editor, it is recommended that the GDExtension's native binaries are compiled not just for Android, but also for the OS onto which developers / users intend to run the Godot Editor. Not doing so may prevent developers / users from writing code that accesses the plugin from within the Godot Editor.

This may involve creating dummy plugins for the host OS just so the API is published to the editor. You can use the godot-cpp-template [https://github.com/godotengine/godot-cpp-template] github template for reference on how to do so.

Godot crashes upon load

Check `adb logcat` for possible problems, then:

- Check that the methods exposed by the plugin used the following Java types: `void`, `boolean`, `int`, `float`, `java.lang.String`, `org.godotengine.godot.Dictionary`, `int[]`, `byte[]`, `float[]`, `java.lang.String[]`.
- More complex datatypes are not supported for now.

Android in-app purchases

Godot offers a first-party `GodotGooglePlayBilling` Android plugin compatible with Godot 4 which uses the Google Play Billing library [https://developer.android.com/google/play/billing].

Usage

Getting started

Make sure you have enabled and successfully set up Android Gradle Builds. Follow the compiling instructions on the `GodotGooglePlayBilling` github page [https://github.com/godotengine/godot-google-play-billing].

Then put the files *./godot-google-play-billing/build/outputs/aar/GodotGooglePlayBilling.***.release.aar* and *./GodotGooglePlayBilling.gdap* into your project in the *res://android/plugins* folder.

The plugin should now show up in the Android export settings, where you can enable it.

Initialize the plugin

To use the `GodotGooglePlayBilling` API:

1. Obtain a reference to the `GodotGooglePlayBilling` singleton
2. Connect handlers for the plugin signals
3. Call `startConnection`

Initialization example:

```
var payment

func _ready():
    if Engine.has_singleton("GodotGooglePlayBilling"):
        payment =
Engine.get_singleton("GodotGooglePlayBilling")

        # These are all signals supported by the API
        # You can drop some of these based on your needs
        payment.billing_resume.connect(_on_billing_resume) # No
params
        payment.connected.connect(_on_connected) # No params
        payment.disconnected.connect(_on_disconnected) # No
params
        payment.connect_error.connect(_on_connect_error) #
Response ID (int), Debug message (string)

payment.price_change_acknowledged.connect(_on_price_acknowledge
d) # Response ID (int)

payment.purchases_updated.connect(_on_purchases_updated) #
Purchases (Dictionary[])
        payment.purchase_error.connect(_on_purchase_error) #
Response ID (int), Debug message (string)

payment.sku_details_query_completed.connect(_on_product_details
_query_completed) # Products (Dictionary[])

payment.sku_details_query_error.connect(_on_product_details_que
ry_error) # Response ID (int), Debug message (string), Queried
SKUs (string[])

payment.purchase_acknowledged.connect(_on_purchase_acknowledged
) # Purchase token (string)

payment.purchase_acknowledgement_error.connect(_on_purchase_ack
nowledgement_error) # Response ID (int), Debug message
(string), Purchase token (string)

payment.purchase_consumed.connect(_on_purchase_consumed) #
Purchase token (string)

payment.purchase_consumption_error.connect(_on_purchase_consump
tion_error) # Response ID (int), Debug message (string),
Purchase token (string)

payment.query_purchases_response.connect(_on_query_purchases_re
sponse) # Purchases (Dictionary[])

        payment.startConnection()
    else:
        print("Android IAP support is not enabled. Make sure
you have enabled 'Gradle Build' and the GodotGooglePlayBilling
plugin in your Android export settings! IAP will not work.")
```

The API must be in a connected state prior to use. The `connected` signal is sent when the connection process succeeds. You can also use `isReady()` to determine if the plugin is ready for use. The `getConnectionState()` function returns the current connection state of the plugin.

Return values for `getConnectionState()`:

```
# Matches BillingClient.ConnectionState in the Play Billing
Library
enum ConnectionState {
    DISCONNECTED, # not yet connected to billing service or was
already closed
    CONNECTING, # currently in process of connecting to billing
service
    CONNECTED, # currently connected to billing service
    CLOSED, # already closed and shouldn't be used again
}
```

Query available items

Once the API has connected, query SKUs using `querySkuDetails()`. You must successfully complete an SKU query before calling the `purchase()` or `queryPurchases()` functions, or they will return an error. `querySkuDetails()` takes two parameters: an array of SKU name strings, and a string specifying the type of SKU being queried.

The SKU type string should be `"inapp"` for normal in-app purchases or `"subs"` for subscriptions. The name strings in the array should match the SKU product ids defined in the Google Play Console entry for your app.

Example use of `querySkuDetails()`:

```
func _on_connected():
  payment.querySkuDetails(["my_iap_item"], "inapp") # "subs" for subscriptions

func _on_product_details_query_completed(product_details):
  for available_product in product_details:
    print(available_product)

func _on_product_details_query_error(response_id, error_message, products_queried):
    print("on_product_details_query_error id:", response_id, " message: ",
            error_message, " products: ", products_queried)
```

Query user purchases

To retrieve a user's purchases, call the `queryPurchases()` function passing a string with the type of SKU to query. The SKU type string should be `"inapp"` for normal in-app purchases or `"subs"` for subscriptions. The `query_purchases_response` signal is sent with the result. The signal has a single parameter: a Dictionary with a status code and either an array of purchases or an error message. Only active subscriptions and non-consumed one-time purchases are included in the purchase array.

Example use of `queryPurchases()`:

```
func _query_purchases():
    payment.queryPurchases("inapp") # Or "subs" for subscriptions
```

```
func _on_query_purchases_response(query_result):
    if query_result.status == OK:
        for purchase in query_result.purchases:
            _process_purchase(purchase)
    else:
        print("queryPurchases failed, response code: ",
                query_result.response_code,
                " debug message: ", query_result.debug_message)
```

You should query purchases during startup after successfully retrieving SKU details. Since the user may make a purchase or resolve a pending transaction from outside your app, you should recheck for purchases when resuming from the background. To accomplish this, you can use the `billing_resume` signal.

Example use of `billing_resume`:

```
func _on_billing_resume():
    if payment.getConnectionState() ==
ConnectionState.CONNECTED:
        _query_purchases()
```

For more information on processing the purchase items returned by `queryPurchases()`, see Processing a purchase item

Purchase an item

To initiate the purchase flow for an item, call `purchase()` passing the product id string of the SKU you wish to purchase. Reminder: you **must** query the SKU details for an item before you can pass it to `purchase()`.

Example use of `purchase()`:

```
payment.purchase("my_iap_item")
```

The payment flow will send a `purchases_updated` signal on success or a `purchase_error` signal on failure.

```
func _on_purchases_updated(purchases):
    for purchase in purchases:
        _process_purchase(purchase)

func _on_purchase_error(response_id, error_message):
    print("purchase_error id:", response_id, " message: ",
error_message)
```

Processing a purchase item

The `query_purchases_response` and `purchases_updated` signals provide an array of purchases in Dictionary format. The purchase Dictionary includes keys that map to values of the Google Play Billing Purchase [https://developer.android.com/reference/com/android/billingclient/api/Purchase] class.

Purchase fields:

```
dictionary.put("order_id", purchase.getOrderId());
dictionary.put("package_name", purchase.getPackageName());
dictionary.put("purchase_state", purchase.getPurchaseState());
dictionary.put("purchase_time", purchase.getPurchaseTime());
dictionary.put("purchase_token", purchase.getPurchaseToken());
dictionary.put("quantity", purchase.getQuantity());
dictionary.put("signature", purchase.getSignature());
// PBL V4 replaced getSku with getSkus to support multi-sku
purchases,
// use the first entry for "sku" and generate an array for
"skus"
ArrayList<String> skus = purchase.getSkus();
dictionary.put("sku", skus.get(0)); # Not available in plugin
String[] skusArray = skus.toArray(new String[0]);
dictionary.put("products", productsArray);
dictionary.put("is_acknowledged", purchase.isAcknowledged());
dictionary.put("is_auto_renewing", purchase.isAutoRenewing());
```

Check purchase state

Check the `purchase_state` value of a purchase to determine if a purchase was completed or is still pending.

PurchaseState values:

```
# Matches Purchase.PurchaseState in the Play Billing Library
enum PurchaseState {
    UNSPECIFIED,
    PURCHASED,
    PENDING,
}
```

If a purchase is in a `PENDING` state, you should not award the contents of the purchase or do any further processing of the purchase until it reaches the `PURCHASED` state. If you have a store interface, you may wish to display information about pending purchases needing to be completed in the Google Play Store. For more details on pending purchases, see [Handling pending transactions](https://developer.android.com/google/play/billing/integrate#pending) [https://developer.android.com/google/play/billing/integrate#pending] in the Google Play Billing Library documentation.

Consumables

If your in-app item is not a one-time purchase but a consumable item (e.g. coins) which can be purchased multiple times, you can consume an item by calling `consumePurchase()` passing the `purchase_token` value from the purchase dictionary. Calling `consumePurchase()` automatically acknowledges a purchase. Consuming a product allows the user to purchase it again, it will no longer appear in subsequent `queryPurchases()` calls unless it is repurchased.

Example use of `consumePurchase()`:

```
func _process_purchase(purchase):
    if "my_consumable_iap_item" in purchase.products and
purchase.purchase_state == PurchaseState.PURCHASED:
        # Add code to store payment so we can reconcile the
purchase token
        # in the completion callback against the original
purchase
        payment.consumePurchase(purchase.purchase_token)

func _on_purchase_consumed(purchase_token):
    _handle_purchase_token(purchase_token, true)

func _on_purchase_consumption_error(response_id, error_message,
purchase_token):
    print("_on_purchase_consumption_error id:", response_id,
            " message: ", error_message)
    _handle_purchase_token(purchase_token, false)

# Find the sku associated with the purchase token and award the
# product if successful
func _handle_purchase_token(purchase_token,
purchase_successful):
    # check/award logic, remove purchase from tracking list
```

Acknowledging purchases

If your in-app item is a one-time purchase, you must acknowledge the purchase by calling the `acknowledgePurchase()` function, passing the `purchase_token` value from the purchase dictionary. If you do not acknowledge a purchase within three days, the user automatically receives a refund, and Google Play revokes the purchase. If you are calling `comsumePurchase()` it automatically acknowledges the purchase and you do not need to call `acknowledgePurchase()`.

Example use of `acknowledgePurchase()`:

```
func _process_purchase(purchase):
    if "my_one_time_iap_item" in purchase.products and \
            purchase.purchase_state == PurchaseState.PURCHASED
and \
            not purchase.is_acknowledged:
        # Add code to store payment so we can reconcile the
purchase token
        # in the completion callback against the original
purchase
        payment.acknowledgePurchase(purchase.purchase_token)

func _on_purchase_acknowledged(purchase_token):
    _handle_purchase_token(purchase_token, true)

func _on_purchase_acknowledgement_error(response_id,
error_message, purchase_token):
    print("_on_purchase_acknowledgement_error id: ",
response_id,
            " message: ", error_message)
    _handle_purchase_token(purchase_token, false)

# Find the sku associated with the purchase token and award the
# product if successful
func _handle_purchase_token(purchase_token,
purchase_successful):
    # check/award logic, remove purchase from tracking list
```

Subscriptions

Subscriptions work mostly like regular in-app items. Use `"subs"` as the second argument to `querySkuDetails()` to get subscription details. Pass `"subs"` to `queryPurchases()` to get subscription purchase details.

You can check `is_auto_renewing` in the a subscription purchase returned from `queryPurchases()` to see if a user has cancelled an auto-renewing subscription.

You need to acknowledge new subscription purchases, but not automatic subscription renewals.

If you support upgrading or downgrading between different subscription levels, you should use `updateSubscription()` to use the subscription update flow to change an active subscription. Like `purchase()`, results are returned by the `purchases_updated` and `purchase_error` signals. There are three parameters to `updateSubscription()`:

1. The purchase token of the currently active subscription
2. The product id string of the subscription SKU to change to
3. The proration mode to apply to the subscription.

The proration values are defined as:

```
enum SubscriptionProrationMode {
    # Replacement takes effect immediately, and the remaining
time
    # will be prorated and credited to the user.
    IMMEDIATE_WITH_TIME_PRORATION = 1,
    # Replacement takes effect immediately, and the billing
cycle remains the same.
    # The price for the remaining period will be charged.
    # This option is only available for subscription upgrade.
    IMMEDIATE_AND_CHARGE_PRORATED_PRICE,
    # Replacement takes effect immediately, and the new price
will be charged on
    # next recurrence time. The billing cycle stays the same.
    IMMEDIATE_WITHOUT_PRORATION,
    # Replacement takes effect when the old plan expires, and
the new price
    # will be charged at the same time.
    DEFERRED,
    # Replacement takes effect immediately, and the user is
charged full price
    # of new plan and is given a full billing cycle of
subscription,
    # plus remaining prorated time from the old plan.
    IMMEDIATE_AND_CHARGE_FULL_PRICE,
}
```

Default behavior is `IMMEDIATE_WITH_TIME_PRORATION`.

Example use of `updateSubscription`:

```
payment.updateSubscription(_active_subscription_purchase.purcha
se_token, \
                    "new_sub_sku",
SubscriptionProrationMode.IMMEDIATE_WITH_TIME_PRORATION)
```

The `confirmPriceChange()` function can be used to launch price change confirmation flow for a subscription. Pass the product id of the subscription SKU subject to the price change. The result will be sent by the `price_change_acknowledged` signal.

Example use of `confirmPriceChange()`:

```
enum BillingResponse {SUCCESS = 0, CANCELLED = 1}

func confirm_price_change(product_id):
    payment.confirmPriceChange(product_id)

func _on_price_acknowledged(response_id):
    if response_id == BillingResponse.SUCCESS:
        print("price_change_accepted")
    elif response_id == BillingResponse.CANCELED:
        print("price_change_canceled")
```

iOS plugins

- Creating iOS plugins
- Plugins for iOS

Creating iOS plugins

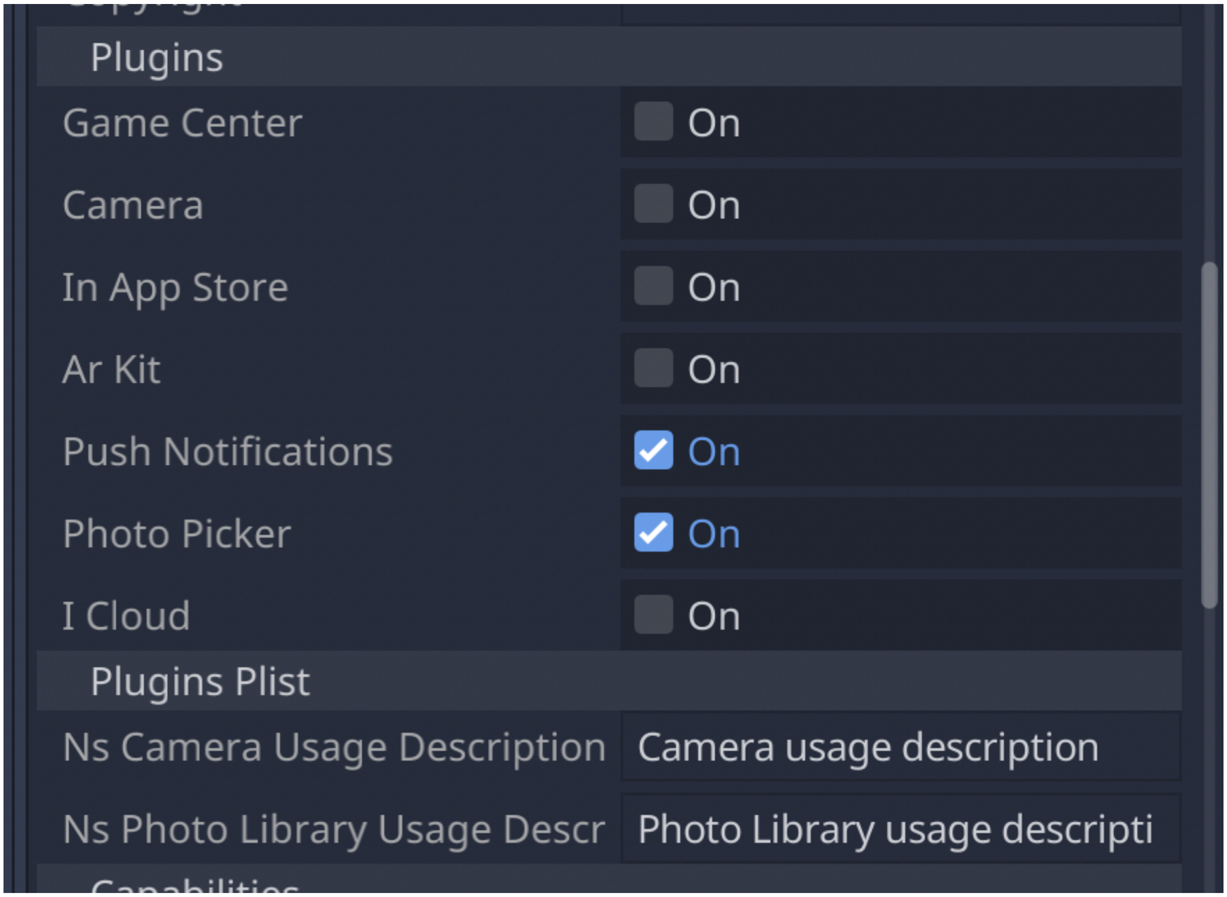
This page explains what iOS plugins can do for you, how to use an existing plugin, and the steps to code a new one.

iOS plugins allow you to use third-party libraries and support iOS-specific features like In-App Purchases, GameCenter integration, ARKit support, and more.

Loading and using an existing plugin

An iOS plugin requires a `.gdip` configuration file, a binary file which can be either `.a` static library or `.xcframework` containing `.a` static libraries, and possibly other dependencies. To use it, you need to:

1. Copy the plugin's files to your Godot project's `res://ios/plugins` directory. You can also group files in a sub-directory, like `res://ios/plugins/my_plugin`.
2. The Godot editor automatically detects and imports `.gdip` files inside `res://ios/plugins` and its subdirectories.
3. You can find and activate detected plugins by going to Project -> Export... -> iOS and in the Options tab, scrolling to the Plugins section.



Plugins

Game Center ☐ On

Camera ☐ On

In App Store ☐ On

Ar Kit ☐ On

Push Notifications ☑ On

Photo Picker ☑ On

I Cloud ☐ On

Plugins Plist

Ns Camera Usage Description: Camera usage description

Ns Photo Library Usage Descr: Photo Library usage descripti

When a plugin is active, you can access it in your using `Engine.get_singleton()`:

```
if Engine.has_singleton("MyPlugin"):
    var singleton = Engine.get_singleton("MyPlugin")
    print(singleton.foo())
```

Note

The plugin's files have to be in the `res://ios/plugins/` directory or a subdirectory, otherwise the Godot editor will not automatically detect them.

Creating an iOS plugin

At its core, a Godot iOS plugin is an iOS library (*.a* archive file or *.xcframework* containing static libraries) with the following requirements:

- The library must have a dependency on the Godot engine headers.
- The library must come with a `.gdip` configuration file.

An iOS plugin can have the same functionality as a Godot module but provides more flexibility and doesn't require to rebuild the engine.

Here are the steps to get a plugin's development started. We recommend using Xcode [https://developer.apple.com/develop/] as your development environment.

See also

The Godot iOS Plugins [https://github.com/godotengine/godot-ios-plugins].

The Godot iOS plugin template [https://github.com/naithar/godot_ios_plugin] gives you all the boilerplate you need to get your iOS plugin started.

To build an iOS plugin:

1. Create an Objective-C static library for your plugin inside Xcode.
2. Add the Godot engine header files as a dependency for your plugin library in `HEADER_SEARCH_PATHS`. You can find the setting inside the `Build Settings` tab:
   - Download the Godot engine source from the Godot GitHub page [https://github.com/godotengine/godot].
   - Run SCons to generate headers. You can learn the process by reading Compiling for iOS. You don't have

    to wait for compilation to complete to move forward as headers are generated before the engine starts to compile.
  - You should use the same header files for iOS plugins and for the iOS export template.

3. In the `Build Settings` tab, specify the compilation flags for your static library in `OTHER_CFLAGS`. The most important ones are `-fcxx-modules`, `-fmodules`, and `-DDEBUG` if you need debug support. Other flags should be the same you use to compile Godot. For instance:

```
-DPTRCALL_ENABLED -DDEBUG_ENABLED -DDEBUG_MEMORY_ALLOC -
DDISABLE_FORCED_INLINE -DTYPED_METHOD_BIND
```

4. Add the required logic for your plugin and build your library to generate a `.a` file. You will probably need to build both `debug` and `release` target `.a` files. Depending on your needs, pick either or both. If you need both debug and release `.a` files, their name should match following pattern: `[PluginName].[TargetType].a`. You can also build the static library with your SCons configuration.
5. The iOS plugin system also supports `.xcframework` files. To generate one, you can use a command such as:

```
xcodebuild -create-xcframework -library [DeviceLibrary].a -
library [SimulatorLibrary].a -output [PluginName].xcframework
```

6. Create a Godot iOS Plugin configuration file to help the system detect and load your plugin:

   - The configuration file extension must be `gdip` (e.g.: `MyPlugin.gdip`).

   - The configuration file format is as follow:

```
[config]
name="MyPlugin"
binary="MyPlugin.a"

initialization="init_my_plugin"
deinitialization="deinit_my_plugin"

[dependencies]
linked=[]
embedded=[]
system=["Foundation.framework"]

capabilities=["arkit", "metal"]

files=["data.json"]

linker_flags=["-ObjC"]

[plist]
PlistKeyWithDefaultType="Some Info.plist key you might need"
StringPlistKey:string="String value"
IntegerPlistKey:integer=42
BooleanPlistKey:boolean=true
RawPlistKey:raw="
<array>
    <string>UIInterfaceOrientationPortrait</string>
</array>
"
StringPlistKeyToInput:string_input="Type something"
```

The `config` section and fields are required and defined as follow:

- **name**: name of the plugin
- **binary**: this should be the filepath of the plugin library (`a` or `xcframework`) file.
  - The filepath can be relative (e.g.: `MyPlugin.a`, `MyPlugin.xcframework`) in which case it's relative to the directory where the `gdip` file is located.

    - The filepath can be absolute: `res://some_path/MyPlugin.a` or `res://some_path/MyPlugin.xcframework`.
    - In case you need multitarget library usage, the filename should be `MyPlugin.a` and `.a` files should be named as `MyPlugin.release.a` and `MyPlugin.debug.a`.
    - In case you use multitarget `xcframework` libraries, their filename in the configuration should be `MyPlugin.xcframework`. The `.xcframework` files should be named as `MyPlugin.release.xcframework` and `MyPlugin.debug.xcframework`.

The `dependencies` and `plist` sections are optional and defined as follow:

- **dependencies**:
    - **linked**: contains a list of iOS frameworks that the iOS application should be linked with.
    - **embedded**: contains a list of iOS frameworks or libraries that should be both linked and embedded into the resulting iOS application.
    - **system**: contains a list of iOS system frameworks that are required for plugin.
    - **capabilities**: contains a list of iOS capabilities that is required for plugin. A list of available capabilities can be found at Apple UIRequiredDeviceCapabilities documentation page [https://developer.apple.com/documentation/bun

dleresources/information_property_list/uireddevicecapabilities].

  - **files**: contains a list of files that should be copied on export. This is useful for data files or images.
  - **linker_flags**: contains a list of linker flags to add to the Xcode project when exporting the plugin.
- **plist**: should have keys and values that should be present in `Info.plist` file.
  - Each line should follow pattern: `KeyName:KeyType=KeyValue`
  - Supported values for `KeyType` are `string`, `integer`, `boolean`, `raw`, `string_input`
  - If no type is used (e.g.: `KeyName="KeyValue"`) `string` type will be used.
  - If `raw` type is used value for corresponding key will be stored in `Info.plist` as is.
  - If `string_input` type is used you will be able to modify value in Export window.

Plugins for iOS

Godot provides StoreKit, GameCenter, iCloud services and other plugins. They are using same model of asynchronous calls explained below.

ARKit and Camera access are also provided as plugins.

Latest updates, documentation and source code can be found at Godot iOS plugins repository [https://github.com/godotengine/godot-ios-plugins]

Accessing plugin singletons

To access plugin functionality, you first need to check that the plugin is exported and available by calling the *Engine.has_singleton()* function, which returns a registered singleton.

Here's an example of how to do this in GDScript:

```
var in_app_store
var game_center

func _ready():
    if Engine.has_singleton("InAppStore"):
        in_app_store = Engine.get_singleton("InAppStore")
    else:
        print("iOS IAP plugin is not available on this platform.")

    if Engine.has_singleton("GameCenter"):
        game_center = Engine.get_singleton("GameCenter")
    else:
        print("iOS Game Center plugin is not available on this platform.")
```

Asynchronous methods

When requesting an asynchronous operation, the method will look like this:

```
Error purchase(Variant params);
```

The parameter will usually be a Dictionary, with the information necessary to make the request, and the call will have two phases. First, the method will immediately return an Error value. If the Error is not 'OK', the call operation is completed, with an error probably caused locally (no internet connection, API incorrectly configured, etc). If the error value is 'OK', a response event will be produced and added to the 'pending events' queue. Example:

```
func on_purchase_pressed():
    var result = in_app_store.purchase({ "product_id": "my_product" })
    if result == OK:
        animation.play("busy") # show the "waiting for response" animation
    else:
        show_error()

# put this on a 1 second timer or something
func check_events():
    while in_app_store.get_pending_event_count() > 0:
        var event = in_app_store.pop_pending_event()
        if event.type == "purchase":
            if event.result == "ok":
                show_success(event.product_id)
            else:
                show_error()
```

Remember that when a call returns OK, the API will *always* produce an event through the pending_event interface, even if it's an error, or a network timeout, etc. You should be able to, for example, safely block the interface waiting for a reply from the server. If any of the APIs don't behave this way it should be treated as a bug.

The pending event interface consists of two methods:

- `get_pending_event_count()` Returns the number of pending events on the queue.
- `Variant pop_pending_event()` Pops the first event from the queue and returns it.

Store Kit

Implemented in Godot iOS InAppStore plugin [https://github.com/godotengine/godot-ios-plugins/blob/master/plugins/inappstore/in_app_store.mm].

The Store Kit API is accessible through the `InAppStore` singleton. It is initialized automatically.

The following methods are available and documented below:

```
   Error purchase(Variant params)
   Error request_product_info(Variant params)
   Error restore_purchases()
   void set_auto_finish_transaction(bool enable)
   void finish_transaction(String product_id)

and the pending events interface:

::

   int get_pending_event_count()
   Variant pop_pending_event()
```

`purchase`

Purchases a product ID through the Store Kit API. You have to call `finish_transaction(product_id)` once you receive a successful response or call `set_auto_finish_transaction(true)` prior to calling `purchase()`. These two methods ensure the transaction is completed.

Parameters

Takes a dictionary as a parameter, with one field, `product_id`, a string with your product ID. Example:

```
var result = in_app_store.purchase({ "product_id": "my_product"
})
```

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "purchase",
  "result": "error",
  "product_id": "the product ID requested",
}
```

On success:

```
{
  "type": "purchase",
  "result": "ok",
  "product_id": "the product ID requested",
}
```

`request_product_info`

Requests the product info on a list of product IDs.

Parameters

Takes a dictionary as a parameter, with a single `product_ids` key to which a string array of product IDs is assigned. Example:

```
var result = in_app_store.request_product_info({ "product_ids":
["my_product1", "my_product2"] })
```

Response event

The response event will be a dictionary with the following fields:

```
{
  "type": "product_info",
  "result": "ok",
  "invalid_ids": [ list of requested IDs that were invalid ],
  "ids": [ list of IDs that were valid ],
  "titles": [ list of valid product titles (corresponds with
list of valid IDs) ],
  "descriptions": [ list of valid product descriptions ],
  "prices": [ list of valid product prices ],
  "localized_prices": [ list of valid product localized prices
],
}
```

`restore_purchases`

Restores previously made purchases on user's account. This will create response events for each previously purchased product ID.

Response event

The response events will be dictionaries with the following fields:

```
{
  "type": "restore",
  "result": "ok",
  "product_id": "product ID of restored purchase",
}
```

`set_auto_finish_transaction`

If set to `true`, once a purchase is successful, your purchase will be finalized automatically. Call this method prior to calling `purchase()`.

Parameters

Takes a boolean as a parameter which specifies if purchases should be automatically finalized. Example:

```
in_app_store.set_auto_finish_transaction(true)
```

`finish_transaction`

If you don't want transactions to be automatically finalized, call this method after you receive a successful purchase response.

Parameters

Takes a string `product_id` as an argument. `product_id` specifies what product to finalize the purchase on. Example:

```
in_app_store.finish_transaction("my_product1")
```

Game Center

Implemented in Godot iOS GameCenter plugin [https://github.com/godotengine/godot-ios-plugins/blob/master/plugins/gamecenter/game_center.mm].

The Game Center API is available through the `GameCenter` singleton. It has the following methods:

```
Error authenticate()
bool is_authenticated()
Error post_score(Variant score)
Error award_achievement(Variant params)
void reset_achievements()
void request_achievements()
void request_achievement_descriptions()
Error show_game_center(Variant params)
Error request_identity_verification_signature()
```

and the pending events interface:

```
int get_pending_event_count()
Variant pop_pending_event()
```

`authenticate`

Authenticates a user in Game Center.

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "authentication",
  "result": "error",
  "error_code": the value from NSError::code,
  "error_description": the value from
NSError::localizedDescription,
}
```

On success:

```
{
  "type": "authentication",
  "result": "ok",
  "player_id": the value from GKLocalPlayer::playerID,
}
```

`post_score`

Posts a score to a Game Center leaderboard.

Parameters

Takes a dictionary as a parameter, with two fields:

- `score` a float number
- `category` a string with the category name

Example:

```
var result = game_center.post_score({ "score": 100, "category":
"my_leaderboard", })
```

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "post_score",
  "result": "error",
  "error_code": the value from NSError::code,
  "error_description": the value from
NSError::localizedDescription,
}
```

On success:

```
{
  "type": "post_score",
  "result": "ok",
}
```

`award_achievement`

Modifies the progress of a Game Center achievement.

Parameters

Takes a Dictionary as a parameter, with 3 fields:

- `name` (string) the achievement name
- `progress` (float) the achievement progress from 0.0 to 100.0 (passed to `GKAchievement::percentComplete`)
- `show_completion_banner` (bool) whether Game Center should display an achievement banner at the top of the screen

Example:

```
var result = award_achievement({ "name": "hard_mode_completed",
"progress": 6.1 })
```

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "award_achievement",
  "result": "error",
  "error_code": the error code taken from NSError::code,
}
```

On success:

```
{
  "type": "award_achievement",
  "result": "ok",
}
```

`reset_achievements`

Clears all Game Center achievements. The function takes no parameters.

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "reset_achievements",
  "result": "error",
  "error_code": the value from NSError::code,
}
```

On success:

```
{
  "type": "reset_achievements",
  "result": "ok",
}
```

`request_achievements`

Request all the Game Center achievements the player has made progress on. The function takes no parameters.

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "achievements",
  "result": "error",
  "error_code": the value from NSError::code,
}
```

On success:

```
{
  "type": "achievements",
  "result": "ok",
  "names": [ list of the name of each achievement ],
  "progress": [ list of the progress made on each achievement
],
}
```

`request_achievement_descriptions`

Request the descriptions of all existing Game Center achievements regardless of progress. The function takes no parameters.

Response event

The response event will be a dictionary with the following fields:

On error:

```
{
  "type": "achievement_descriptions",
  "result": "error",
  "error_code": the value from NSError::code,
}
```

On success:

```
{
  "type": "achievement_descriptions",
  "result": "ok",
  "names": [ list of the name of each achievement ],
  "titles": [ list of the title of each achievement ],
  "unachieved_descriptions": [ list of the description of each
achievement when it is unachieved ],
  "achieved_descriptions": [ list of the description of each
achievement when it is achieved ],
  "maximum_points": [ list of the points earned by completing
each achievement ],
  "hidden": [ list of booleans indicating whether each
achievement is initially visible ],
  "replayable": [ list of booleans indicating whether each
achievement can be earned more than once ],
}
```

show_game_center

Displays the built in Game Center overlay showing leaderboards, achievements, and challenges.

Parameters

Takes a Dictionary as a parameter, with two fields:

- `view` (string) (optional) the name of the view to present. Accepts "default", "leaderboards", "achievements", or "challenges". Defaults to "default".
- `leaderboard_name` (string) (optional) the name of the leaderboard to present. Only used when "view" is "leaderboards" (or "default" is configured to show leaderboards). If not specified, Game Center will display the aggregate leaderboard.

Examples:

```
var result = show_game_center({ "view": "leaderboards",
"leaderboard_name": "best_time_leaderboard" })
```

```
var result = show_game_center({ "view": "achievements" })
```

Response event

The response event will be a dictionary with the following fields:

On close:

```
{
  "type": "show_game_center",
  "result": "ok",
}
```

Web

- The JavaScriptBridge Singleton
- HTML5 shell class reference
- Custom HTML page for Web export

The JavaScriptBridge Singleton

In web builds, the JavaScriptBridge singleton allows interaction with JavaScript and web browsers, and can be used to implement some functionalities unique to the web platform.

Interacting with JavaScript

Sometimes, when exporting Godot for the Web, it might be necessary to interface with external JavaScript code like third-party SDKs, libraries, or simply to access browser features that are not directly exposed by Godot.

The `JavaScriptBridge` singleton provides methods to wrap a native JavaScript object into a Godot JavaScriptObject that tries to feel natural in the context of Godot scripting (e.g. GDScript and C#).

The JavaScriptBridge.get_interface() method retrieves an object in the global scope.

```
extends Node

func _ready():
    # Retrieve the `window.console` object.
    var console = JavaScriptBridge.get_interface("console")
    # Call the `window.console.log()` method.
    console.log("test")
```

The JavaScriptBridge.create_object() creates a new object via the JavaScript `new` constructor.

```
extends Node
```

```
func _ready():
    # Call the JavaScript `new` operator on the `window.Array`
object.
    # Passing 10 as argument to the constructor:
    # JS: `new Array(10);`
    var arr = JavaScriptBridge.create_object("Array", 10)
    # Set the first element of the JavaScript array to the
number 42.
    arr[0] = 42
    # Call the `pop` function on the JavaScript array.
    arr.pop()
    # Print the value of the `length` property of the array (9
after the pop).
    print(arr.length)
```

As you can see, by wrapping JavaScript objects into `JavaScriptObject` you can interact with them like they were native Godot objects, calling their methods, and retrieving (or even setting) their properties.

Base types (int, floats, strings, booleans) are automatically converted (floats might lose precision when converted from Godot to JavaScript). Anything else (i.e. objects, arrays, functions) are seen as `JavaScriptObjects` themselves.

Callbacks

Calling JavaScript code from Godot is nice, but sometimes you need to call a Godot function from JavaScript instead.

This case is a bit more complicated. JavaScript relies on garbage collection, while Godot uses reference counting for memory management. This means you have to explicitly create callbacks (which are returned as `JavaScriptObjects` themselves) and you have to keep their reference.

Arguments passed by JavaScript to the callback will be passed as a single Godot `Array`.

```
extends Node

# Here we create a reference to the `_my_callback` function
(below).
# This reference will be kept until the node is freed.
var _callback_ref =
JavaScriptBridge.create_callback(_my_callback)

func _ready():
    # Get the JavaScript `window` object.
    var window = JavaScriptBridge.get_interface("window")
    # Set the `window.onbeforeunload` DOM event listener.
    window.onbeforeunload = _callback_ref

func _my_callback(args):
    # Get the first argument (the DOM event in our case).
    var js_event = args[0]
    # Call preventDefault and set the `returnValue` property of
the DOM event.
    js_event.preventDefault()
    js_event.returnValue = ''
```

Here is another example that asks the user for the [Notification permission](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API) [https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API] and waits asynchronously to deliver a notification if the permission is granted:

```
extends Node

# Here we create a reference to the `_on_permissions` function
(below).
# This reference will be kept until the node is freed.
var _permission_callback =
JavaScriptBridge.create_callback(_on_permissions)

func _ready():
    # NOTE: This is done in `_ready` for simplicity, but SHOULD
BE done in response
```

```
    # to user input instead (e.g. during `_input`, or `button_pressed` event, etc.),
    # otherwise it might not work.

    # Get the `window.Notification` JavaScript object.
    var notification = JavaScriptBridge.get_interface("Notification")
    # Call the `window.Notification.requestPermission` method which returns a JavaScript
    # Promise, and bind our callback to it.
    notification.requestPermission().then(_permission_callback)

func _on_permissions(args):
    # The first argument of this callback is the string "granted" if the permission is granted.
    var permission = args[0]
    if permission == "granted":
        print("Permission granted, sending notification.")
        # Create the notification: `new Notification("Hi there!")`
        JavaScriptBridge.create_object("Notification", "Hi there!")
    else:
        print("No notification permission.")
```

Can I use my favorite library?

You most likely can. First, you have to include your library in the page. You can simply customize the Head Include during export (see below), or even write your own template.

In the example below, we customize the `Head Include` to add an external library (axios [https://axios-http.com/]) from a content delivery network, and a second `<script>` tag to define our own custom function:

```
<!-- Axios -->
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
```

```
</script>
<!-- Custom function -->
<script>
function myFunc() {
    alert("My func!");
}
</script>
```

We can then access both the library and the function from Godot, like we did in previous examples:

```
extends Node

# Here create a reference to the `_on_get` function (below).
# This reference will be kept until the node is freed.
var _callback = JavaScriptBridge.create_callback(_on_get)

func _ready():
    # Get the `window` object, where globally defined functions
are.
    var window = JavaScriptBridge.get_interface("window")
    # Call the JavaScript `myFunc` function defined in the
custom HTML head.
    window.myFunc()
    # Get the `axios` library (loaded from a CDN in the custom
HTML head).
    var axios = JavaScriptBridge.get_interface("axios")
    # Make a GET request to the current location, and receive
the callback when done.
    axios.get(window.location.toString()).then(_callback)

func _on_get(args):
    OS.alert("On Get")
```

The eval interface

The `eval` method works similarly to the JavaScript function of the same name. It takes a string as an argument and executes it as

JavaScript code. This allows interacting with the browser in ways not possible with script languages integrated into Godot.

GDScript

```
func my_func():
    JavaScriptBridge.eval("alert('Calling JavaScript per
GDScript!');")
```

C#

```
private void MyFunc()
{
    JavaScriptBridge.Eval("alert('Calling JavaScript per
C#!');")
}
```

The value of the last JavaScript statement is converted to a GDScript value and returned by `eval()` under certain circumstances:

- JavaScript `number` is returned as float
- JavaScript `boolean` is returned as bool
- JavaScript `string` is returned as String
- JavaScript `ArrayBuffer`, `TypedArray`, and `DataView` are returned as PackedByteArray

GDScript

```
func my_func2():
    var js_return = JavaScriptBridge.eval("var myNumber = 1;
myNumber + 2;")
    print(js_return) # prints '3.0'
```

C#

```
private void MyFunc2()
{
    var jsReturn = JavaScriptBridge.Eval("var myNumber = 1;
myNumber + 2;");
```

```
    GD.Print(jsReturn); // prints '3.0'
}
```

Any other JavaScript value is returned as `null`.

HTML5 export templates may be built without support for the singleton to improve security. With such templates, and on platforms other than HTML5, calling `JavaScriptBridge.eval` will also return `null`. The availability of the singleton can be checked with the `web` feature tag:

GDScript

```
func my_func3():
    if OS.has_feature('web'):
        JavaScriptBridge.eval("""
            console.log('The JavaScriptBridge singleton is
available')
        """)
    else:
        print("The JavaScriptBridge singleton is NOT
available")
```

C#

```
private void MyFunc3()
{
    if (OS.HasFeature("web"))
    {
        JavaScriptBridge.Eval("console.log('The
JavaScriptBridge singleton is available')");
    }
    else
    {
        GD.Print("The JavaScriptBridge singleton is NOT
available");
    }
}
```

Tip

GDScript's multi-line strings, surrounded by 3 quotes `"""` as in `my_func3()` above, are useful to keep JavaScript code readable.

The `eval` method also accepts a second, optional Boolean argument, which specifies whether to execute the code in the global execution context, defaulting to `false` to prevent polluting the global namespace:

GDScript

```
func my_func4():
    # execute in global execution context,
    # thus adding a new JavaScript global variable `SomeGlobal`
    JavaScriptBridge.eval("var SomeGlobal = {};", true)
```

C#

```
private void MyFunc4()
{
    // execute in global execution context,
    // thus adding a new JavaScript global variable
`SomeGlobal`
    JavaScriptBridge.Eval("var SomeGlobal = {};", true);
}
```

Downloading files

Downloading files (e.g. a save game) from the Godot Web export to the user's computer can be done by directly interacting with JavaScript, but given it is a very common use case, Godot exposes this functionality to scripting via a dedicated JavaScriptBridge.download_buffer() function which lets you download any generated buffer.

Here is a minimal example on how to use it:

extends Node

```
func _ready():
    # Asks the user download a file called "hello.txt" whose content will be the string "Hello".
    JavaScriptBridge.download_buffer("Hello".to_utf8_buffer(), "hello.txt")
```

And here is a more complete example on how to download a previously saved file:

```
extends Node

# Open a file for reading and download it via the JavaScript singleton.
func _download_file(path):
    var file = FileAccess.open(path, FileAccess.READ)
    if file == null:
        push_error("Failed to load file")
        return
    # Get the file name.
    var fname = path.get_file()
    # Read the whole file to memory.
    var buffer = file.get_buffer(file.get_len())
    # Prompt the user to download the file (will have the same name as the input file).
    JavaScriptBridge.download_buffer(buffer, fname)

func _ready():
    # Create a temporary file.
    var config = ConfigFile.new()
    config.set_value("option", "one", false)
    config.save("/tmp/test.cfg")

    # Download it
    _download_file("/tmp/test.cfg")
```

HTML5 shell class reference

Projects exported for the Web expose the `Engine()` class to the JavaScript environment, that allows fine control over the engine's start-up process.

This API is built in an asynchronous manner and requires basic understanding of Promises [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises].

Engine

The `Engine` class provides methods for loading and starting exported projects on the Web. For default export settings, this is already part of the exported HTML page. To understand practical use of the `Engine` class, see Custom HTML page for Web export.

Static Methods

Promise	`load` **(** string basePath **)**
void	`unload` **( )**
boolean	`isWebGLAvailable` **(** *[number majorVersion=1]* **)**

Instance Methods

Promise	`init` **(** *[string basePath]* **)**

Promise	`preloadFile` **(** string|ArrayBuffer file *[, string path]* **)**
Promise	`start` **(** EngineConfig override **)**
Promise	`startGame` **(** EngineConfig override **)**
void	`copyToFS` **(** string path, ArrayBuffer buffer **)**
void	`requestQuit` **( )**

class Engine(*initConfig*)

Create a new Engine instance with the given configuration.

Arguments:
- **initConfig** (`EngineConfig()`) -- The initial config for this instance.

Static Methods

Engine.load(*basePath*)

Load the engine from the specified base path.

Arguments:
- **basePath** (`string()`) -- Base path of the engine to load.

Returns: A Promise that resolves once the engine is loaded.

Return type: Promise

Engine.unload()

Unload the engine to free memory.

This method will be called automatically depending on the configuration. See `unloadAfterInit`.

Engine.isWebGLAvailable([*majorVersion=1*])

Check whether WebGL is available. Optionally, specify a particular version of WebGL to check for.

Arguments:	- **majorVersion** (`number()`) -- The major WebGL version to check for.
Returns:	If the given major version of WebGL is available.
Return type:	boolean

Instance Methods

Engine.prototype.init([*basePath*])

Initialize the engine instance. Optionally, pass the base path to the engine to load it, if it hasn't been loaded yet. See `Engine.load()`.

Arguments:	- **basePath** (`string()`) -- Base path of the engine to load.
Returns:	A `Promise` that resolves once the engine is loaded and initialized.
Return type:	Promise

Engine.prototype.preloadFile(*file*[, *path*])

Load a file so it is available in the instance's file system once it runs. Must be called **before** starting the instance.

If not provided, the `path` is derived from the URL of the loaded file.

Arguments:

- **file** (`string|ArrayBuffer()`) -- The file to preload.
  If a `string` the file will be loaded from tha path.
  If an `ArrayBuffer` or a view on one, the buffer will used as the content of the file.
- **path** (`string()`) -- Path by which the file will be accessible. Required, if `file` is not string.

Returns: A Promise that resolves once the file is loaded.

Return type: Promise

Engine.prototype.start(*override*)

Start the engine instance using the given override configuration (if any). `startGame` can be used in typical cases instead.

This will initialize the instance if it is not initialized. For manual initialization, see `init`. The engine must be loaded beforehand.

Fails if a canvas cannot be found on the page, or not specified in the configuration.

Arguments:

- **override** (`EngineConfig()`) -- An optional configuration override.

Returns: Promise that resolves once the engine started.

Return type: Promise

Engine.prototype.startGame(*override*)

Start the game instance using the given configuration override (if any).

This will initialize the instance if it is not initialized. For manual initialization, see `init`.

This will load the engine if it is not loaded, and preload the main pck.

This method expects the initial config (or the override) to have both the `executable` and `mainPack` properties set (normally done by the editor during export).

Arguments:	- **override** (`EngineConfig()`) -- An optional configuration override.
Returns:	Promise that resolves once the game started.
Return type:	Promise

Engine.prototype.copyToFS(*path*, *buffer*)

Create a file at the specified `path` with the passed as `buffer` in the instance's file system.

Arguments:	- **path** (`string()`) -- The location where the file will be created. - **buffer** (`ArrayBuffer()`) -- The content of the file.

Engine.prototype.requestQuit()

Request that the current instance quit.

This is akin the user pressing the close button in the window manager, and will have no effect if the engine has crashed, or is stuck in a loop.

Engine configuration

An object used to configure the Engine instance based on godot export options, and to override those in custom HTML templates if needed.

Properties

type	name
boolean	`unloadAfterInit`
HTMLCanvasElement	`canvas`
string	`executable`
string	`mainPack`
string	`locale`
number	`canvasResizePolicy`
Array.<string>	`args`
function	`onExecute`
function	`onExit`
function	`onProgress`
function	`onPrint`

function	`onPrintError`

EngineConfig

The Engine configuration object. This is just a typedef, create it like a regular object, e.g.:

```
const MyConfig = { executable: 'godot', unloadAfterInit: false }
```

Property Descriptions

unloadAfterInit

Whether the unload the engine automatically after the instance is initialized.

Type: boolean
Value: `true`

canvas

The HTML DOM Canvas object to use.

By default, the first canvas element in the document will be used is none is specified.

Type: HTMLCanvasElement
Value: `null`

executable

The name of the WASM file without the extension. (Set by Godot Editor export process).

Type: string
Value: `""`

mainPack

An alternative name for the game pck to load. The executable name is used otherwise.

Type:	string
Value:	`null`

locale

Specify a language code to select the proper localization for the game.

The browser locale will be used if none is specified. See complete list of supported locales.

Type:	string
Value:	`null`

canvasResizePolicy

The canvas resize policy determines how the canvas should be resized by Godot.

`0` means Godot won't do any resizing. This is useful if you want to control the canvas size from javascript code in your template.

`1` means Godot will resize the canvas on start, and when changing window size via engine functions.

`2` means Godot will adapt the canvas size to match the whole browser window.

Type:	number
Value:	`2`

args

The arguments to be passed as command line arguments on startup.

See command line tutorial.

Note: `startGame` will always add the `--main-pack` argument.

Type:	Array.<string>
Value:	`[]`

onExecute(*path*, *args*)

A callback function for handling Godot's `OS.execute` calls.

This is for example used in the Web Editor template to switch between Project Manager and editor, and for running the game.

Arguments:

- **path** (`string()`) -- The path that Godot's wants executed.
- **args** (`Array.`) -- The arguments of the "command" to execute.

onExit(*status_code*)

A callback function for being notified when the Godot instance quits.

Note: This function will not be called if the engine crashes or become unresponsive.

Arguments:

- **status_code** (`number()`) -- The status code returned by Godot on exit.

onProgress(*current*, *total*)

A callback function for displaying download progress.

The function is called once per frame while downloading files, so the usage of `requestAnimationFrame()` is not necessary.

If the callback function receives a total amount of bytes as 0, this means that it is impossible to calculate. Possible reasons include:

- Files are delivered with server-side chunked compression

- Files are delivered with server-side compression on Chromium
- Not all file downloads have started yet (usually on servers without multi-threading)

Arguments:

- **current** (`number()`) -- The current amoun of downloaded bytes so far.
- **total** (`number()`) -- The total amount of bytes to be downloaded.

onPrint([*...var_args*])

A callback function for handling the standard output stream. This method should usually only be used in debug pages.

By default, `console.log()` is used.

Arguments:

- **var_args** (`*()`) -- A variadic number of arguments to be printed.

onPrintError([*...var_args*])

A callback function for handling the standard error stream. This method should usually only be used in debug pages.

By default, `console.error()` is used.

Arguments:

- **var_args** (`*()`) -- A variadic number of arguments to be printed as errors.

Custom HTML page for Web export

While Web export templates provide a default HTML page fully capable of launching the project without any further customization, it may be beneficial to create a custom HTML page. While the game itself cannot easily be directly controlled from the outside yet, such page allows to customize the initialization process for the engine.

Some use-cases where customizing the default page is useful include:

- Loading files from a different directory than the page;
- Loading a `.zip` file instead of a `.pck` file as the main pack;
- Loading the engine from a different directory than the main pack file;
- Adding a click-to-play button so that games can be started in the fullscreen mode;
- Loading some extra files before the engine starts, making them available in the project file system as soon as possible;
- Passing custom command line arguments, e.g. `-s` to start a `MainLoop` script.

The default HTML page is available in the Godot Engine repository at /misc/dist/html/full-size.html [https://github.com/godotengine/godot/blob/master/misc/dist/html/full-size.html] but the following template can be used as a much simpler example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Template</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <canvas id="canvas"></canvas>
        <script src="$GODOT_URL"></script>
        <script>
            var engine = new Engine($GODOT_CONFIG);
            engine.startGame();
        </script>
    </body>
</html>
```

Setup

As shown by the example above, it is mostly a regular HTML document, with few placeholders which needs to be replaced during export, an html `<canvas>` element, and some simple JavaScript code that calls the **`Engine()`** class.

The only required placeholders are:

- `$GODOT_URL`: The name of the main JavaScript file, which provides the **`Engine()`** class required to start the engine and that must be included in the HTML as a `<script>`. The name is generated from the *Export Path* during the export process.
- `$GODOT_CONFIG`: A JavaScript object, containing the export options and can be later overridden. See **`EngineConfig`** for the full list of overrides.

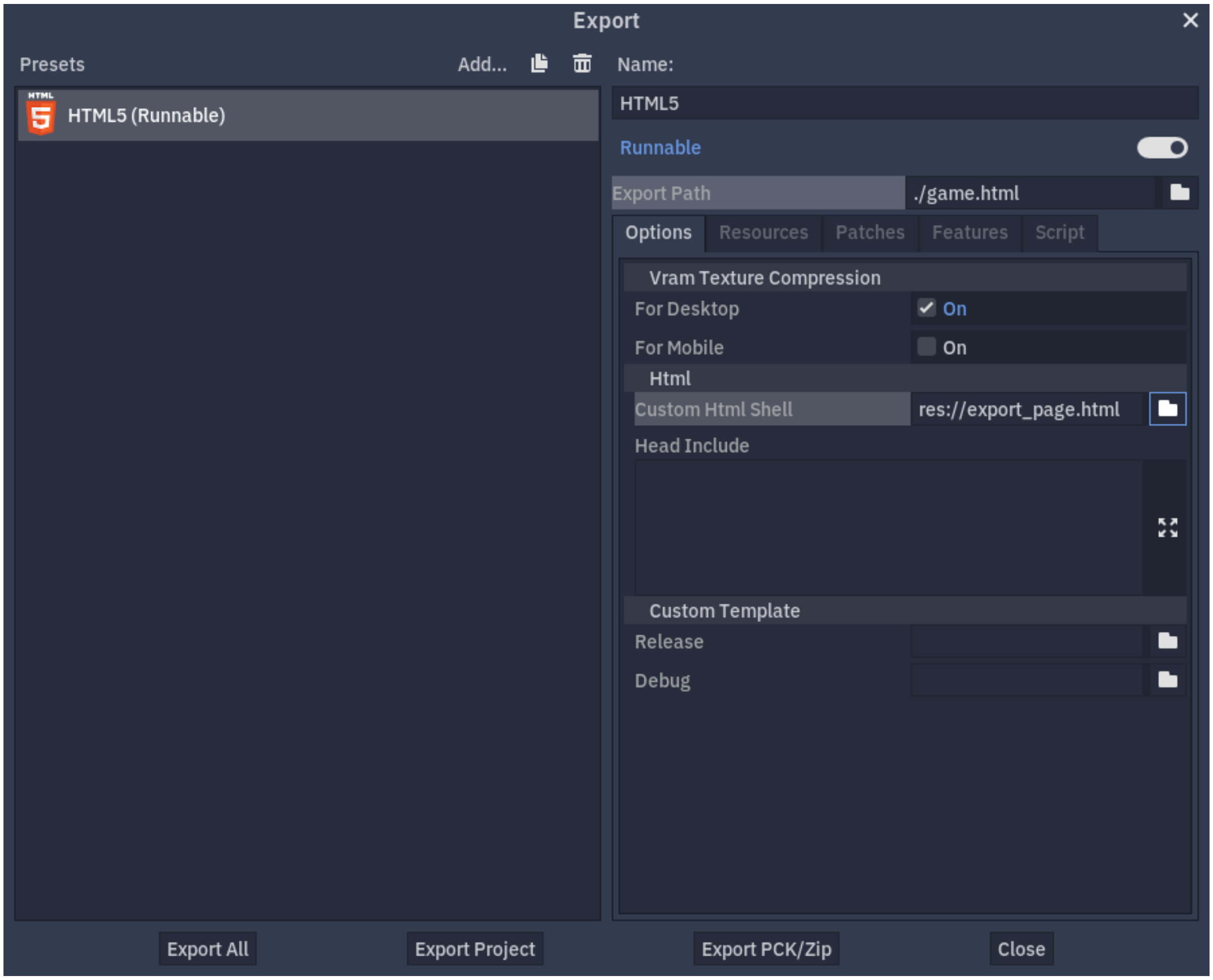
The following optional placeholders will enable some extra features in your custom HTML template.

- `$GODOT_PROJECT_NAME`: The project name as defined in the Project Settings. It is a good idea to use it as a `<title>` in your template.
- `$GODOT_HEAD_INCLUDE`: A custom string to include in the HTML document just before the end of the `<head>` tag. It is customized in the export options under the *Html / Head Include* section.

While you fully control the HTML page you create, this variable can be useful for configuring parts of the HTML `head` element from the Godot Editor, e.g. for different Web export presets.

- `$GODOT_SPLASH`: Path to an image used as the boot splash as defined in the Project Settings.

When the custom page is ready, it can be selected in the export options under the *Html / Custom Html Shell* section.



Name: HTML5

Runnable: ●

Export Path: ./game.html

Vram Texture Compression

For Desktop: ☑ On

For Mobile: ☐ On

Html

Custom Html Shell: res://export_page.html

Head Include:

Custom Template

Release:

Debug:

Export
Presets
Add...
HTML5 (Runnable)
Options
Resources
Patches
Features
Script
Export All
Export Project
Export PCK/Zip
Close

Starting the project

To be able to start the game, you need to write a script that initializes the engine — the control code. This process consists of three steps, but as shown here, most of them can be skipped depending on how much customization is needed.

See the HTML5 shell class reference, for the full list of methods and options available.

First, the engine must be loaded, then it needs to be initialized, and after this the project can finally be started. You can perform every of these steps manually and with great control. However, in the simplest case all you need to do is to create an instance of the `Engine()` class with the exported configuration, and then call the `engine.startGame` method optionally overriding any `EngineConfig` parameters.

```
const engine = new Engine($GODOT_CONFIG);
engine.startGame({
    /* optional override configuration, eg. */
    // unloadAfterInit: false,
    // canvasResizePolicy: 0,
    // ...
});
```

This snippet of code automatically loads and initializes the engine before starting the game. It uses the given configuration to load the engine. The `engine.startGame` method is asynchronous and returns a `Promise`. This allows your control code to track if the game was loaded correctly without blocking execution or relying on polling.

In case your project needs to have special control over the start arguments and dependency files, the `engine.start` method can be used instead. Note, that this method do not automatically preload the `pck` file, so you will probably want to manually preload it (and any other extra file) via the `engine.preloadFile` method.

Optionally, you can also manually `engine.init` to perform specific actions after the module initialization, but before the engine starts.

This process is a bit more complex, but gives you full control over the engine startup process.

```
const myWasm = 'mygame.wasm';
const myPck = 'mygame.pck';
const engine = new Engine();
Promise.all([
    // Load and init the engine
    engine.init(myWasm),
    // And the pck concurrently
    engine.preloadFile(myPck),
]).then(() => {
    // Now start the engine.
    return engine.start({ args: ['--main-pack', myPck] });
}).then(() => {
    console.log('Engine has started!');
});
```

To load the engine manually the `Engine.load()` static method must be called. As this method is static, multiple engine instances can be spawned if the share the same `wasm`.

Note

Multiple instances cannot be spawned by default, as the engine is immediately unloaded after it is initialized. To prevent this from happening see the `unloadAfterInit` override option. It is still possible to unload the engine manually afterwards by calling the `Engine.unload()` static method. Unloading the engine frees browser memory by unloading files that are no longer needed once the instance is initialized.

Customizing the behavior

In the Web environment several methods can be used to guarantee that the game will work as intended.

If you target a specific version of WebGL, or just want to check if WebGL is available at all, you can call the `Engine.isWebGLAvailable()` method. It optionally takes an argument that allows to test for a specific major version of WebGL.

As the real executable file does not exist in the Web environment, the engine only stores a virtual filename formed from the base name of loaded engine files. This value affects the output of the OS.get_executable_path() method and defines the name of the automatically started main pack. The `executable` override option can be used to override this value.

Customizing the presentation

Several configuration options can be used to further customize the look and behavior of the game on your page.

By default, the first canvas element on the page is used for rendering. To use a different canvas element the `canvas` override option can be used. It requires a reference to the DOM element itself.

```
const canvasElement = document.querySelector("#my-canvas-
element");
engine.startGame({ canvas: canvasElement });
```

The way the engine resize the canvas can be configured via the `canvasResizePolicy` override option.

If your game takes some time to load, it may be useful to display a custom loading UI which tracks the progress. This can be achieved with the `onProgress` callback option, which allows to set up a callback function that will be called regularly as the engine loads new bytes.

```
function printProgress(current, total) {
    console.log("Loaded " + current + " of " + total + " bytes");
}
engine.startGame({ onProgress: printProgress });
```

Be aware that in some cases `total` can be `0`. This means that it cannot be calculated.

If your game supports multiple languages, the **`locale`** override option can be used to force a specific locale, provided you have a valid language code string. It may be good to use server-side logic to determine which languages a user may prefer. This way the language code can be taken from the `Accept-Language` HTTP header, or determined by a GeoIP service.

Debugging

To debug exported projects, it may be useful to read the standard output and error streams generated by the engine. This is similar to the output shown in the editor console window. By default, standard `console.log` and `console.warn` are used for the output and error streams respectively. This behavior can be customized by setting your own functions to handle messages.

Use the **`onPrint`** override option to set a callback function for the output stream, and the **`onPrintError`** override option to set a callback function for the error stream.

```
function print(text) {
    console.log(text);
}
function printError(text) {
    console.warn(text);
}
engine.startGame({ onPrint: print, onPrintError: printError });
```

When handling the engine output, keep in mind that it may not be desirable to print it out in the finished product.

Console support in Godot

Console porting process

In order to develop for consoles in Godot, you need access to the console SDK and export templates for it. These export templates need to be developed either by yourself or someone hired to do it, or provided by a third party company.

Currently, the only console Godot officially supports is Steam Deck (through the official Linux export templates).

The reason other consoles are not officially supported are:

- To develop for consoles, one must be licensed as a company. As an open source project, Godot has no legal structure to provide console ports.
- Console SDKs are secret and covered by non-disclosure agreements. Even if we could get access to them, we could not publish the platform-specific code under an open source license.

As explained, however, it is possible to port your games to consoles thanks to services provided by third-party companies.

Note

In practice, the process is quite similar to Unity and Unreal Engine. In other words, there is no engine that is legally allowed to distribute console export templates without requiring the user to prove that they are a licensed console developer.

Console publishing process

Regardless of the engine used to create the game, the process to publish a game to a console platform is as follows:

- Register a developer account on the console manufacturer's website, then sign NDAs and publishing contracts. This requires you to have a registered legal entity.
- Gain access to the publishing platform by passing the acceptance process. This can take up to several months. Note that this step is significantly easier if an established publisher is backing your game. Nintendo is generally known to be more accepting of smaller developers, but this is not guaranteed.
- Get access to developer tools and order a console specially made for developers (*devkit*). The cost of those devkits is confidential.
- Port the engine to the console platform or pay a company to do it.
- To be published, your game needs to be rated in the regions you'd like to sell it in. For example, game ratings are handled by ESRB [https://www.esrb.org/] in North America, and PEGI [https://pegi.info/] in Europe. Indie developers can generally get a rating for cheaper compared to more established developers.

Due to the complexity of the process, many studios and developers prefer to outsource console porting.

You can read more about the console publishing process in this article: Godot and consoles, all you need to know [https://godotengine.org/article/godot-consoles-all-you-need-know/]

Third-party support

Console ports of Godot are offered by third-party companies (which have ported Godot on their own). These companies also offer publishing of your games to various consoles.

Following is the list of providers:

- Lone Wolf Technology [https://www.lonewolftechnology.com/] offers Switch and Playstation 4 porting and publishing of Godot games.
- Pineapple Works [https://pineapple.works/] offers Switch, Xbox One & Xbox Series X/S (GDK) porting and publishing of Godot games (GDScript/C#).
- RAWRLAB games [https://www.rawrlab.com/] offers Switch porting of Godot games.
- mazette! games [https://mazette.games/] offers Switch, Xbox One and Xbox Series X/S porting and publishing of Godot games.
- Olde Sküül [https://oldeskuul.com/] offers Switch, Xbox One, Playstation 4 & Playstation 5 porting and publishing of Godot games.
- Tuanisapps [https://www.tuanisapps.com/] offers Switch porting and publishing of Godot games.
- Seaven Studio [https://www.seaven-studio.com/] offers Switch, Xbox One, Xbox Series, PlayStation 4 & PlayStation 5 porting of Godot games.
- Sickhead Games [https://www.sickhead.com] offers console porting to Nintendo Switch, PlayStation 4, PlayStation 5, Xbox One, and Xbox Series X/S for Godot games.

If your company offers porting, or porting *and* publishing services for Godot games, feel free to contact the Godot Foundation [https://godot.foundation/#contact] to add your company to the list above.

Plugins

- Editor plugins
- Running code in the editor

Editor plugins

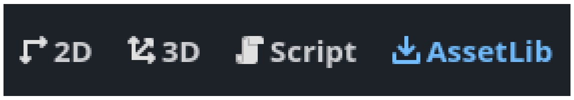
- Installing plugins
- Making plugins
- Making main screen plugins
- Import plugins
- 3D gizmo plugins
- Inspector plugins
- Visual Shader plugins

Installing plugins

Godot features an editor plugin system with numerous plugins developed by the community. Plugins can extend the editor's functionality with new nodes, additional docks, convenience features, and more.

Finding plugins

The preferred way to find Godot plugins is to use the Asset Library [https://godotengine.org/asset-library/]. While it can be browsed online, it's more convenient to use it directly from the editor. To do so, click the **AssetLib** tab at the top of the editor:



You can also find assets on code hosting websites such as GitHub.

Note

Some repositories describe themselves as "plugins" but may not actually be *editor* plugins. This is especially the case for scripts that are intended to be used in a running project. You don't need to enable such plugins to use them. Download them and extract the files in your project folder.

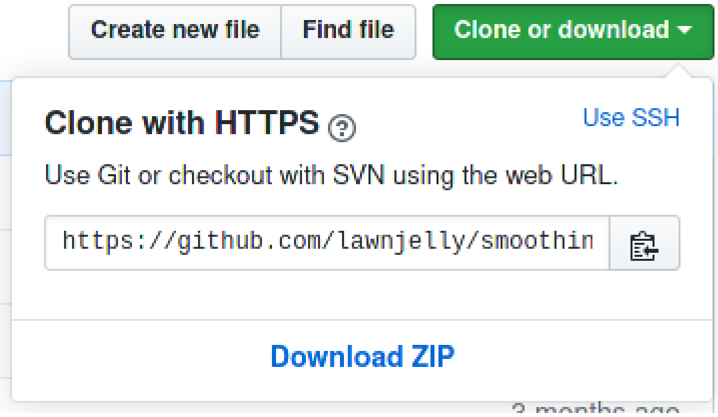
One way to distinguish editor plugins from non-editor plugins is to look for a `plugin.cfg` file in the repository that hosts the plugin. If the repository contains a `plugin.cfg` file in a folder placed in the `addons/` folder, then it is an editor plugin.

Installing a plugin

To install a plugin, download it as a ZIP archive. On the Asset Library, this can be done using the **Download** button, either from the editor or using the Web interface.

On GitHub, if a plugin has *tags* (versions) declared, go to the **Releases** tab to download a stable release. This ensures you download a version that was declared to be stable by its author.

On GitHub, if the plugin doesn't have any *tags* declared, use the **Download ZIP** button to download a ZIP of the latest revision:



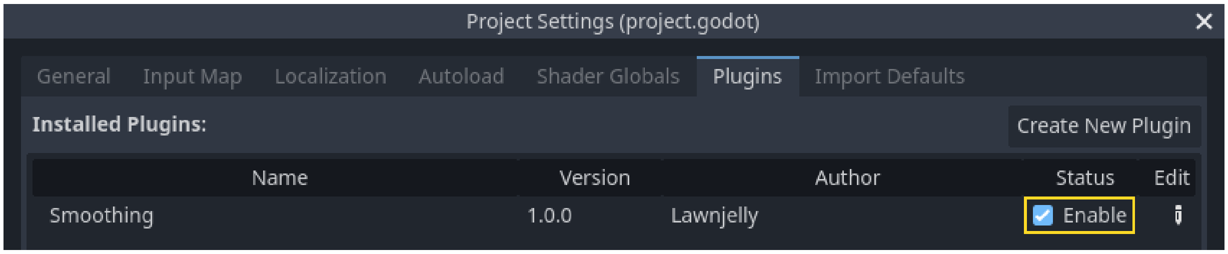
Use Git or checkout with SVN using the web URL.

`https://github.com/lawnjelly/smoothin`

Extract the ZIP archive and move the `addons/` folder it contains into your project folder. If your project already contains an `addons/` folder, move the plugin's `addons/` folder into your project folder to merge the new folder contents with the existing one. Your file manager may ask you whether to write into the folder; answer **Yes**. No files will be overwritten in the process.

Enabling a plugin

To enable the freshly installed plugin, open **Project > Project Settings** at the top of the editor then go the **Plugins** tab. If the plugin was packaged correctly, you should see it in the list of plugins. Click on the **Enable** checkbox to enable the plugin.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Create New Plugin
Status
Edit
☑ Enable

Installed Plugins:

Name	Version	Author
Smoothing	1.0.0	Lawnjelly

You can use the plugin immediately after enabling it; there's no need to restart the editor. Likewise, disabling a plugin can be done without having to restart the editor.

Making plugins

About plugins

A plugin is a great way to extend the editor with useful tools. It can be made entirely with GDScript and standard scenes, without even reloading the editor. Unlike modules, you don't need to create C++ code nor recompile the engine. While this makes plugins less powerful, there are still many things you can do with them. Note that a plugin is similar to any scene you can already make, except it is created using a script to add editor functionality.

This tutorial will guide you through the creation of two plugins so you can understand how they work and be able to develop your own. The first is a custom node that you can add to any scene in the project, and the other is a custom dock added to the editor.

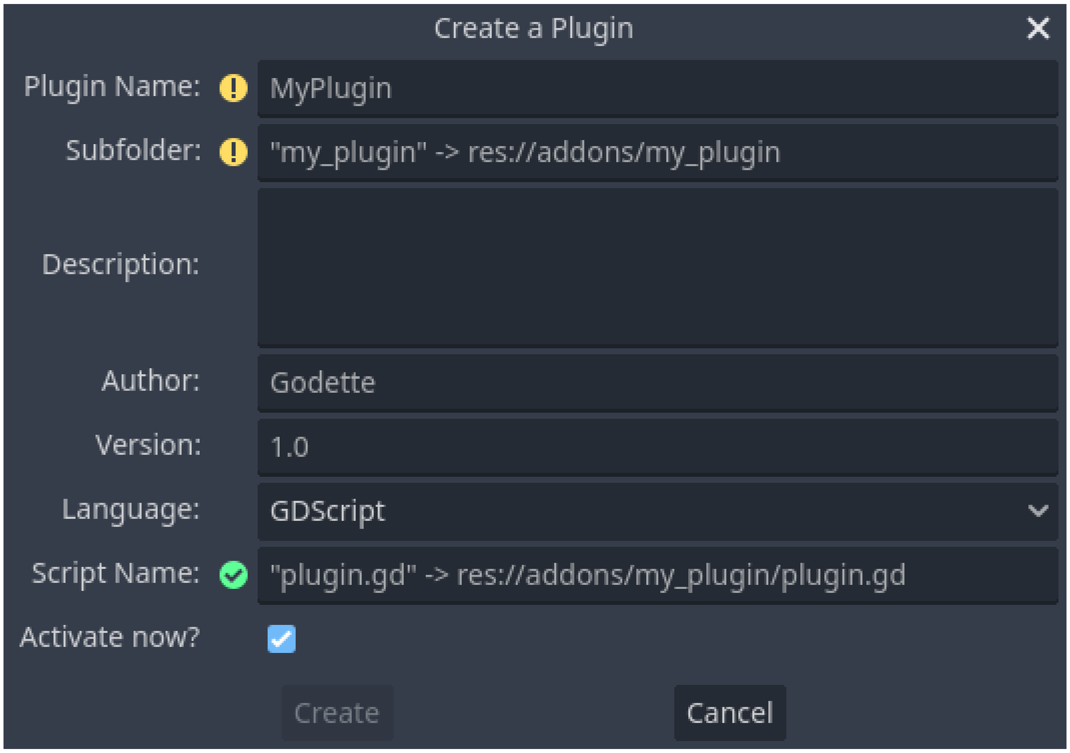
Creating a plugin

Before starting, create a new empty project wherever you want. This will serve as a base to develop and test the plugins.

The first thing you need for the editor to identify a new plugin is to create two files: a `plugin.cfg` for configuration and a tool script with the functionality. Plugins have a standard path like `addons/plugin_name` inside the project folder. Godot provides a dialog for generating those files and placing them where they need to be.

In the main toolbar, click the `Project` dropdown. Then click `Project Settings...`. Go to the `Plugins` tab and then click on the `Create New Plugin` button in the top-right.

You will see the dialog appear, like so:



Plugin Name: MyPlugin
Subfolder: "my_plugin" -> res://addons/my_plugin
Description:
Author: Godette
Version: 1.0
Language: GDScript
Script Name: "plugin.gd" -> res://addons/my_plugin/plugin.gd
Activate now? ☑

The placeholder text in each field describes how it affects the plugin's creation of the files and the config file's values.

To continue with the example, use the following values:

GDScript

```
Plugin Name: My Custom Node
Subfolder: my_custom_node
Description: A custom node made to extend the Godot Engine.
Author: Your Name Here
Version: 1.0.0
Language: GDScript
Script Name: custom_node.gd
Activate now: No
```

C#

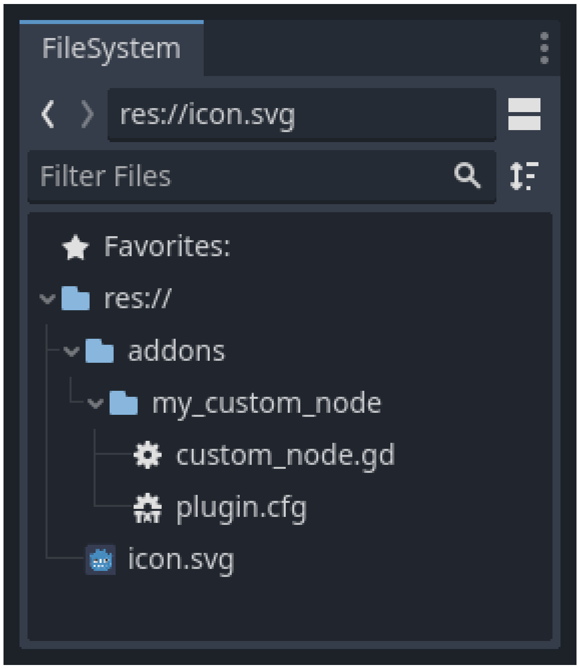
```
Plugin Name: My Custom Node
Subfolder: MyCustomNode
```

```
Description: A custom node made to extend the Godot Engine.
Author: Your Name Here
Version: 1.0.0
Language: C#
Script Name: CustomNode.cs
Activate now: No
```

Warning

Unchecking the `Activate now?` option in C# is always required because, like every other C# script, the EditorPlugin script needs to be compiled which requires building the project. After building the project the plugin can be enabled in the `Plugins` tab of `Project Settings`.

You should end up with a directory structure like this:


FileSystem
res://icon.svg
Filter Files
Favorites:
res://
addons
my_custom_node
custom_node.gd
plugin.cfg
icon.svg

`plugin.cfg` is an INI file with metadata about your plugin. The name and description help people understand what it does. Your name helps you get properly credited for your work. The version number helps others know if they have an outdated version; if you are unsure on how to come up with the version number, check out Semantic Versioning [https://semver.org/]. The main script file will instruct Godot what your plugin does in the editor once it is active.

The script file

Upon creation of the plugin, the dialog will automatically open the EditorPlugin script for you. The script has two requirements that you cannot change: it must be a `@tool` script, or else it will not load properly in the editor, and it must inherit from EditorPlugin.

Warning

In addition to the EditorPlugin script, any other GDScript that your plugin uses must *also* be a tool. Any GDScript without `@tool` used by the editor will act like an empty file!

It's important to deal with initialization and clean-up of resources. A good practice is to use the virtual function _enter_tree() to initialize your plugin and _exit_tree() to clean it up. Thankfully, the dialog generates these callbacks for you. Your script should look something like this:

GDScript

```
@tool
extends EditorPlugin


func _enter_tree():
    # Initialization of the plugin goes here.
    pass


func _exit_tree():
    # Clean-up of the plugin goes here.
    pass

```

C#

```
#if TOOLS
using Godot;

[Tool]
public partial class CustomNode : EditorPlugin
{
    public override void _EnterTree()
    {
        // Initialization of the plugin goes here.
    }

    public override void _ExitTree()
    {
        // Clean-up of the plugin goes here.
    }
}
#endif
```

This is a good template to use when creating new plugins.

A custom node

Sometimes you want a certain behavior in many nodes, such as a custom scene or control that can be reused. Instancing is helpful in a lot of cases, but sometimes it can be cumbersome, especially if you're using it in many projects. A good solution to this is to make a plugin that adds a node with a custom behavior.

Warning

Nodes added via an EditorPlugin are "CustomType" nodes. While they work with any scripting language, they have fewer features than the Script Class system. If you are writing GDScript or NativeScript, we recommend using Script Classes instead.

To create a new node type, you can use the function add_custom_type() from the EditorPlugin class. This function can add new types to the editor (nodes or resources). However, before you can create the type, you need a script that will act as the logic for the type. While that script doesn't have to use the `@tool` annotation, it can be added so the script runs in the editor.

For this tutorial, we'll create a button that prints a message when clicked. For that, we'll need a script that extends from Button. It could also extend BaseButton if you prefer:

GDScript

```
@tool
extends Button


func _enter_tree():
    pressed.connect(clicked)


func clicked():
    print("You clicked me!")
```

C#

```
using Godot;

[Tool]
public partial class MyButton : Button
{
    public override void _EnterTree()
    {
        Pressed += Clicked;
    }

    public void Clicked()
    {
        GD.Print("You clicked me!");
    }
}
```

That's it for our basic button. You can save this as `my_button.gd` inside the plugin folder. You'll also need a 16×16 icon to show in the scene tree. If you don't have one, you can grab the default one from the engine and save it in your *addons/my_custom_node* folder as *icon.png*, or use the default Godot logo (*preload("res://icon.svg")*).

Tip

SVG images that are used as custom node icons should have the **Editor > Scale With Editor Scale** and **Editor > Convert Colors With Editor Theme** import options enabled. This allows icons to follow the editor's scale and theming settings if the icons are designed with the same color palette as Godot's own icons.

Now, we need to add it as a custom type so it shows on the **Create New Node** dialog. For that, change the `custom_node.gd` script to the following:

GDScript

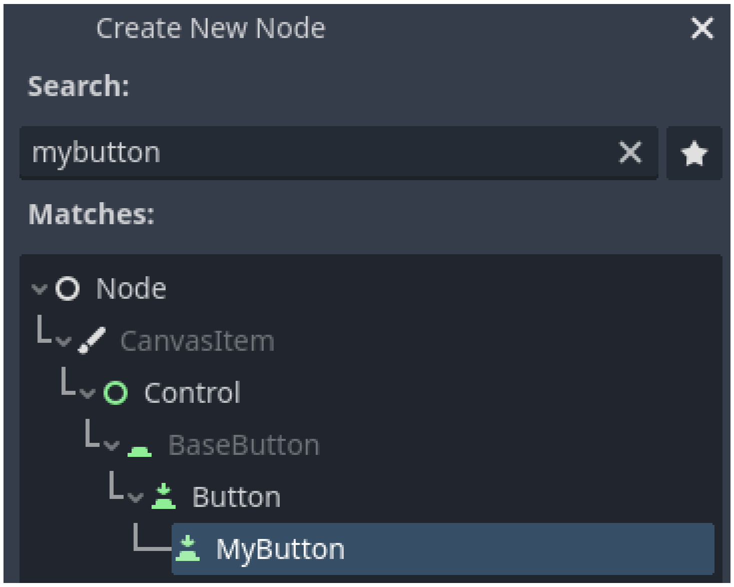
```
@tool
extends EditorPlugin


func _enter_tree():
    # Initialization of the plugin goes here.
    # Add the new type with a name, a parent type, a script and
an icon.
    add_custom_type("MyButton", "Button",
preload("my_button.gd"), preload("icon.png"))


func _exit_tree():
    # Clean-up of the plugin goes here.
    # Always remember to remove it from the engine when
deactivated.
    remove_custom_type("MyButton")
```

C#

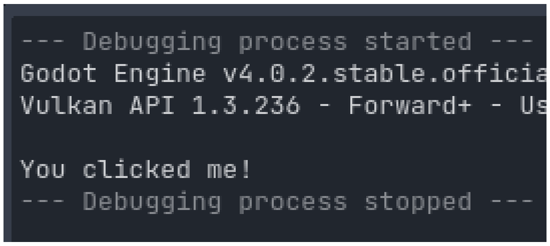
```
#if TOOLS
using Godot;

[Tool]
public partial class CustomNode : EditorPlugin
{
    public override void _EnterTree()
    {
        // Initialization of the plugin goes here.
        // Add the new type with a name, a parent type, a
script and an icon.
        var script = GD.Load<Script>
("res://addons/MyCustomNode/MyButton.cs");
        var texture = GD.Load<Texture2D>
("res://addons/MyCustomNode/Icon.png");
        AddCustomType("MyButton", "Button", script, texture);
    }

    public override void _ExitTree()
    {
        // Clean-up of the plugin goes here.
        // Always remember to remove it from the engine when
deactivated.
        RemoveCustomType("MyButton");
    }
}
#endif
```

With that done, the plugin should already be available in the plugin list in the **Project Settings**, so activate it as explained in Checking the results.

Then try it out by adding your new node:


Create New Node
Search:
mybutton
Matches:
Node
CanvasItem
Control
BaseButton
Button
MyButton

When you add the node, you can see that it already has the script you created attached to it. Set a text to the button, save and run the scene. When you click the button, you can see some text in the console:



```
--- Debugging process started ---
Godot Engine v4.0.2.stable.officia
Vulkan API 1.3.236 - Forward+ - Us

You clicked me!
--- Debugging process stopped ---
```

A custom dock

Sometimes, you need to extend the editor and add tools that are always available. An easy way to do it is to add a new dock with a plugin. Docks are just scenes based on Control, so they are created in a way similar to usual GUI scenes.

Creating a custom dock is done just like a custom node. Create a new `plugin.cfg` file in the `addons/my_custom_dock` folder, then add the following content to it:

GDScript

```
[plugin]

name="My Custom Dock"
description="A custom dock made so I can learn how to make
plugins."
author="Your Name Here"
version="1.0"
script="custom_dock.gd"
```

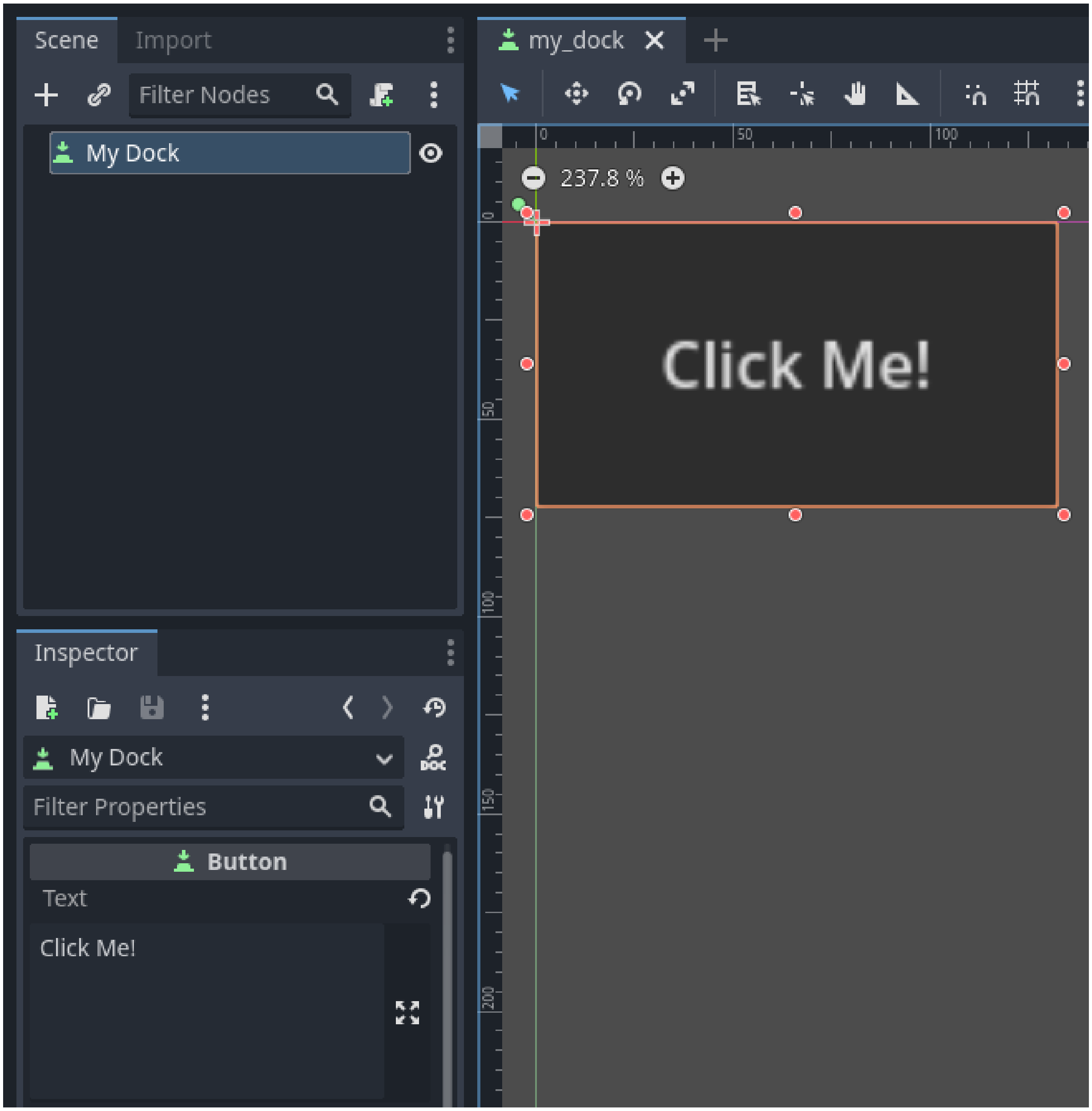
C#

```
[plugin]

name="My Custom Dock"
description="A custom dock made so I can learn how to make
plugins."
author="Your Name Here"
version="1.0"
script="CustomDock.cs"
```

Then create the script `custom_dock.gd` in the same folder. Fill it with the template we've seen before to get a good start.

Since we're trying to add a new custom dock, we need to create the contents of the dock. This is nothing more than a standard Godot scene: just create a new scene in the editor then edit it.

For an editor dock, the root node **must** be a Control or one of its child classes. For this tutorial, you can create a single button. The name of the root node will also be the name that appears on the dock tab, so be sure to give it a short and descriptive name. Also, don't forget to add some text to your button.


Scene
Import
Filter Nodes
My Dock
my_dock
237.8 %
Click Me!
Inspector
My Dock
Filter Properties
Button
Text
Click Me!

Save this scene as `my_dock.tscn`. Now, we need to grab the scene we created then add it as a dock in the editor. For this, you can rely on the function add_control_to_dock() from the EditorPlugin class.

You need to select a dock position and define the control to add (which is the scene you just created). Don't forget to **remove the dock** when the plugin is deactivated. The script could look like this:

GDScript

```
@tool
extends EditorPlugin


# A class member to hold the dock during the plugin life cycle.
var dock


func _enter_tree():
    # Initialization of the plugin goes here.
    # Load the dock scene and instantiate it.
    dock =
preload("res://addons/my_custom_dock/my_dock.tscn").instantiate
()

    # Add the loaded scene to the docks.
    add_control_to_dock(DOCK_SLOT_LEFT_UL, dock)
    # Note that LEFT_UL means the left of the editor, upper-
left dock.


func _exit_tree():
    # Clean-up of the plugin goes here.
    # Remove the dock.
    remove_control_from_docks(dock)
    # Erase the control from the memory.
    dock.free()
```

C#

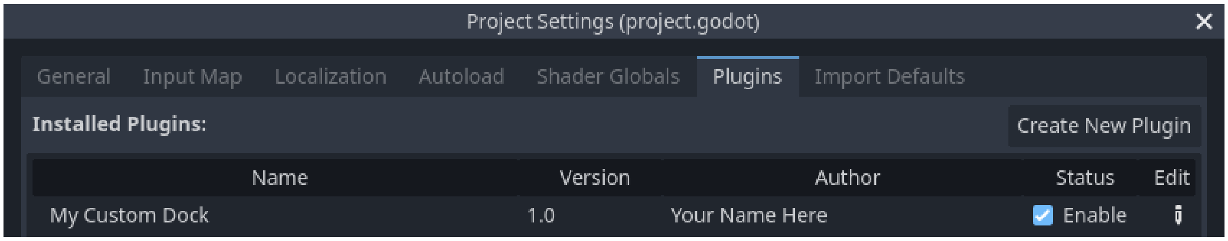
```
#if TOOLS
using Godot;

[Tool]
public partial class CustomDock : EditorPlugin
{
    private Control _dock;

    public override void _EnterTree()
    {
        _dock = GD.Load<PackedScene>
("res://addons/MyCustomDock/MyDock.tscn").Instantiate<Control>
();
```

```
        AddControlToDock(DockSlot.LeftUl, _dock);
    }

    public override void _ExitTree()
    {
        // Clean-up of the plugin goes here.
        // Remove the dock.
        RemoveControlFromDocks(_dock);
        // Erase the control from the memory.
        _dock.Free();
    }
}
#endif
```

Note that, while the dock will initially appear at its specified position, the user can freely change its position and save the resulting layout.

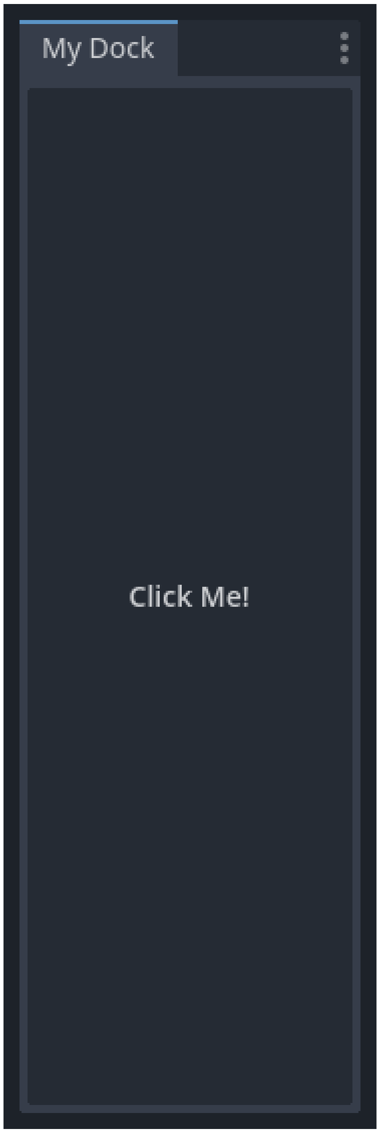
Checking the results

It's now time to check the results of your work. Open the **Project Settings** and click on the **Plugins** tab. Your plugin should be the only one on the list.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Installed Plugins:
Create New Plugin

Name	Version	Author	Status	Edit
My Custom Dock	1.0	Your Name Here	☑ Enable	

You can see the plugin is not enabled. Click the **Enable** checkbox to activate the plugin. The dock should become visible before you even close the settings window. You should now have a custom dock:



Going beyond

Now that you've learned how to make basic plugins, you can extend the editor in several ways. Lots of functionality can be added to the editor with GDScript; it is a powerful way to create specialized editors without having to delve into C++ modules.

You can make your own plugins to help yourself and share them in the Asset Library [https://godotengine.org/asset-library/] so that people can benefit from your work.

Registering autoloads/singletons in plugins

It is possible for editor plugins to automatically register autoloads when the plugin is enabled. This also includes unregistering the autoload when the plugin is disabled.

This makes setting up plugins faster for users, as they no longer have to manually add autoloads to their project settings if your editor plugin requires the use of an autoload.

Use the following code to register a singleton from an editor plugin:

GDScript

```
@tool
extends EditorPlugin

# Replace this value with a PascalCase autoload name, as per
the GDScript style guide.
const AUTOLOAD_NAME = "SomeAutoload"


func _enter_tree():
    # The autoload can be a scene or script file.
    add_autoload_singleton(AUTOLOAD_NAME,
"res://addons/my_addon/some_autoload.tscn")


func _exit_tree():
    remove_autoload_singleton(AUTOLOAD_NAME)
```

C#

```
#if TOOLS
using Godot;

[Tool]
public partial class MyEditorPlugin : EditorPlugin
```

```
{
    // Replace this value with a PascalCase autoload name.
    private const string AutoloadName = "SomeAutoload";

    public override void _EnterTree()
    {
        // The autoload can be a scene or script file.
        AddAutoloadSingleton(AutoloadName,
"res://addons/MyAddon/SomeAutoload.tscn");
    }

    public override void _ExitTree()
    {
        RemoveAutoloadSingleton(AutoloadName);
    }
}
#endif
```

Making main screen plugins

What this tutorial covers

Main screen plugins allow you to create new UIs in the central part of the editor, which appear next to the "2D", "3D", "Script", and "AssetLib" buttons. Such editor plugins are referred as "Main screen plugins".

This tutorial leads you through the creation of a basic main screen plugin. For the sake of simplicity, our main screen plugin will contain a single button that prints text to the console.

Initializing the plugin

First create a new plugin from the Plugins menu. For this tutorial, we'll put it in a folder called `main_screen`, but you can use any name you'd like.

The plugin script will come with `_enter_tree()` and `_exit_tree()` methods, but for a main screen plugin we need to add a few extra methods. Add five extra methods such that the script looks like this:

GDScript

```
@tool
extends EditorPlugin


func _enter_tree():
    pass


func _exit_tree():
    pass


func _has_main_screen():
    return true


func _make_visible(visible):
    pass


func _get_plugin_name():
    return "Main Screen Plugin"


func _get_plugin_icon():
    return EditorInterface.get_editor_theme().get_icon("Node",
"EditorIcons")
```

C#

```
#if TOOLS
using Godot;

[Tool]
public partial class MainScreenPlugin : EditorPlugin
{
    public override void _EnterTree()
    {

    }

    public override void _ExitTree()
    {

    }

    public override bool _HasMainScreen()
    {
        return true;
    }
    public override void _MakeVisible(bool visible)
    {

    }

    public override string _GetPluginName()
    {
        return "Main Screen Plugin";
    }

    public override Texture2D _GetPluginIcon()
    {
        return EditorInterface.GetEditorTheme().GetIcon("Node",
"EditorIcons");
    }
}
#endif
```

The important part in this script is the `_has_main_screen()` function, which is overloaded so it returns `true`. This function is automatically called by the editor on plugin activation, to tell it that this plugin adds a new center view to the editor. For now, we'll leave this script as-is and we'll come back to it later.

Main screen scene

Create a new scene with a root node derived from `Control` (for this example plugin, we'll make the root node a `CenterContainer`). Select this root node, and in the viewport, click the `Layout` menu and select `Full Rect`. You also need to enable the `Expand` vertical size flag in the inspector. The panel now uses all the space available in the main viewport.

Next, let's add a button to our example main screen plugin. Add a `Button` node, and set the text to "Print Hello" or similar. Add a script to the button like this:

GDScript

```
@tool
extends Button


func _on_print_hello_pressed():
    print("Hello from the main screen plugin!")
```

C#

```
using Godot;

[Tool]
public partial class PrintHello : Button
{
    private void OnPrintHelloPressed()
    {
        GD.Print("Hello from the main screen plugin!");
    }
}
```

Then connect the "pressed" signal to itself. If you need help with signals, see the Using signals article.

We are done with the main screen panel. Save the scene as `main_panel.tscn`.

Update the plugin script

We need to update the `main_screen_plugin.gd` script so the plugin instances our main panel scene and places it where it needs to be. Here is the full plugin script:

GDScript

```
@tool
extends EditorPlugin

```

```
const MainPanel = preload("res://addons/main_screen/main_panel.tscn")

var main_panel_instance


func _enter_tree():
    main_panel_instance = MainPanel.instantiate()
    # Add the main panel to the editor's main viewport.

EditorInterface.get_editor_main_screen().add_child(main_panel_instance)
    # Hide the main panel. Very much required.
    _make_visible(false)


func _exit_tree():
    if main_panel_instance:
        main_panel_instance.queue_free()


func _has_main_screen():
    return true


func _make_visible(visible):
    if main_panel_instance:
        main_panel_instance.visible = visible


func _get_plugin_name():
    return "Main Screen Plugin"


func _get_plugin_icon():
    # Must return some kind of Texture for the icon.
    return EditorInterface.get_editor_theme().get_icon("Node", "EditorIcons")
```

C#

```
#if TOOLS
using Godot;

[Tool]
public partial class MainScreenPlugin : EditorPlugin
{
    PackedScene MainPanel = ResourceLoader.Load<PackedScene>
("res://addons/main_screen/main_panel.tscn");
    Control MainPanelInstance;

    public override void _EnterTree()
    {
        MainPanelInstance = (Control)MainPanel.Instantiate();
        // Add the main panel to the editor's main viewport.

EditorInterface.GetEditorMainScreen().AddChild(MainPanelInstanc
e);
        // Hide the main panel. Very much required.
        _MakeVisible(false);
    }

    public override void _ExitTree()
    {
        if (MainPanelInstance != null)
        {
            MainPanelInstance.QueueFree();
        }
    }

    public override bool _HasMainScreen()
    {
        return true;
    }

    public override void _MakeVisible(bool visible)
    {
        if (MainPanelInstance != null)
        {
            MainPanelInstance.Visible = visible;
        }
    }

    public override string _GetPluginName()
    {
        return "Main Screen Plugin";
    }

    public override Texture2D _GetPluginIcon()
    {
        // Must return some kind of Texture for the icon.
        return EditorInterface.GetEditorTheme().GetIcon("Node",
"EditorIcons");
    }
}
#endif
```

A couple of specific lines were added. `MainPanel` is a constant that holds a reference to the scene, and we instance it into *main_panel_instance*.

The `_enter_tree()` function is called before `_ready()`. This is where we instance the main panel scene, and add them as children of specific parts of the editor. We use `EditorInterface.get_editor_main_screen()` to obtain the main editor screen and add our main panel instance as a child to it. We call the `_make_visible(false)` function to hide the main panel so it doesn't compete for space when first activating the plugin.

The `_exit_tree()` function is called when the plugin is deactivated. If the main screen still exists, we call `queue_free()` to free the instance and remove it from memory.

The `_make_visible()` function is overridden to hide or show the main panel as needed. This function is automatically called by the editor when the user clicks on the main viewport buttons at the top of the editor.

The `_get_plugin_name()` and `_get_plugin_icon()` functions control the displayed name and icon for the plugin's main viewport button.

Another function you can add is the `handles()` function, which allows you to handle a node type, automatically focusing the main screen when the type is selected. This is similar to how clicking on a 3D node will automatically switch to the 3D viewport.

Try the plugin

Activate the plugin in the Project Settings. You'll observe a new button next to 2D, 3D, Script above the main viewport. Clicking it will take you to your new main screen plugin, and the button in the middle will print text.

If you would like to try a finished version of this plugin, check out the plugin demos here: https://github.com/godotengine/godot-demo-projects/tree/master/plugins

If you would like to see a more complete example of what main screen plugins are capable of, check out the 2.5D demo projects here: https://github.com/godotengine/godot-demo-projects/tree/master/misc/2.5d

Import plugins

Note

This tutorial assumes you already know how to make generic plugins. If in doubt, refer to the Making plugins page. This also assumes you are acquainted with Godot's import system.

Introduction

An import plugin is a special type of editor tool that allows custom resources to be imported by Godot and be treated as first-class resources. The editor itself comes bundled with a lot of import plugins to handle the common resources like PNG images, Collada and glTF models, Ogg Vorbis sounds, and many more.

This tutorial shows how to create an import plugin to load a custom text file as a material resource. This text file will contain three numeric values separated by comma, which represents the three channels of a color, and the resulting color will be used as the albedo (main color) of the imported material. In this example it contains the pure blue color (zero red, zero green, and full blue):

```
0,0,255
```

Configuration

First we need a generic plugin that will handle the initialization and destruction of our import plugin. Let's add the `plugin.cfg` file first:

```
[plugin]

name="Silly Material Importer"
description="Imports a 3D Material from an external text file."
author="Yours Truly"
version="1.0"
script="material_import.gd"
```

Then we need the `material_import.gd` file to add and remove the import plugin when needed:

```
# material_import.gd
@tool
extends EditorPlugin


var import_plugin


func _enter_tree():
    import_plugin = preload("import_plugin.gd").new()
    add_import_plugin(import_plugin)


func _exit_tree():
    remove_import_plugin(import_plugin)
    import_plugin = null
```

When this plugin is activated, it will create a new instance of the import plugin (which we'll soon make) and add it to the editor using the add_import_plugin() method. We store a reference to it in a class member `import_plugin` so we can refer to it later when removing it. The remove_import_plugin() method is called when the plugin is deactivated to clean up the memory and let the editor know the import plugin isn't available anymore.

Note that the import plugin is a reference type, so it doesn't need to be explicitly released from memory with the `free()` function. It will be released automatically by the engine when it goes out of scope.

The EditorImportPlugin class

The main character of the show is the EditorImportPlugin class. It is responsible for implementing the methods that are called by Godot when it needs to know how to deal with files.

Let's begin to code our plugin, one method at time:

```
# import_plugin.gd
@tool
extends EditorImportPlugin


func _get_importer_name():
    return "demos.sillymaterial"
```

The first method is the _get_importer_name(). This is a unique name for your plugin that is used by Godot to know which import was used in a certain file. When the files needs to be reimported, the editor will know which plugin to call.

```
func _get_visible_name():
    return "Silly Material"
```

The _get_visible_name() method is responsible for returning the name of the type it imports and it will be shown to the user in the Import dock.

You should choose this name as a continuation to "Import as", e.g. *"Import as Silly Material"*. You can name it whatever you want but we recommend a descriptive name for your plugin.

```
func _get_recognized_extensions():
    return ["mtxt"]
```

Godot's import system detects file types by their extension. In the _get_recognized_extensions() method you return an array of strings to represent each extension that this plugin can understand. If an extension is recognized by more than one plugin, the user can select which one to use when importing the files.

Tip

Common extensions like `.json` and `.txt` might be used by many plugins. Also, there could be files in the project that are just data for the game and should not be imported. You have to be careful when importing to validate the data. Never expect the file to be well-formed.

```
func _get_save_extension():
    return "material"
```

The imported files are saved in the `.import` folder at the project's root. Their extension should match the type of resource you are importing, but since Godot can't tell what you'll use (because there might be multiple valid extensions for the same resource), you need to declare what will be used in the import.

Since we're importing a Material, we'll use the special extension for such resource types. If you are importing a scene, you can use `scn`. Generic resources can use the `res` extension. However, this is not enforced in any way by the engine.

```
func _get_resource_type():
    return "StandardMaterial3D"
```

The imported resource has a specific type, so the editor can know which property slot it belongs to. This allows drag and drop from the FileSystem dock to a property in the Inspector.

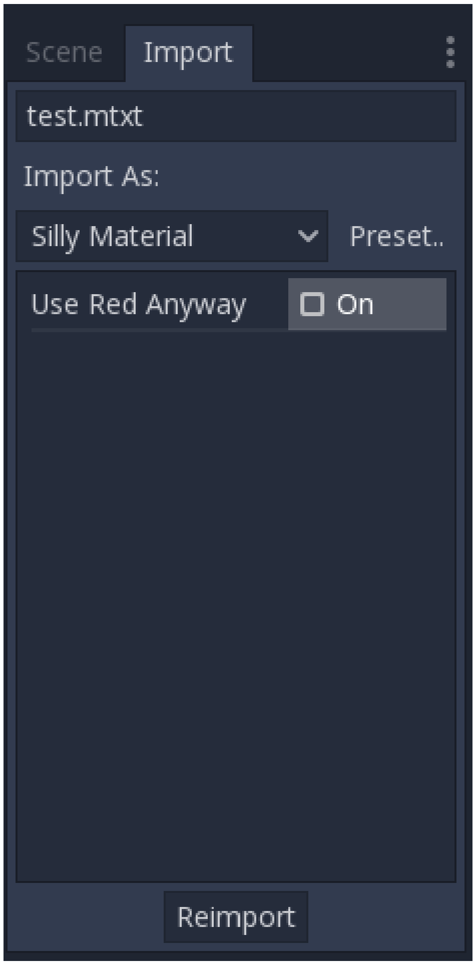
In our case it's a StandardMaterial3D, which can be applied to 3D objects.

Note

If you need to import different types from the same extension, you have to create multiple import plugins. You can abstract the import code on another file to avoid duplication in this regard.

Options and presets

Your plugin can provide different options to allow the user to control how the resource will be imported. If a set of selected options is common, you can also create different presets to make it easier for the user. The following image shows how the options will appear in the editor:



test.mtxt

Use Red Anyway ☐ On

Since there might be many presets and they are identified with a number, it's a good practice to use an enum so you can refer to them using names.

```
@tool
extends EditorImportPlugin


enum Presets { DEFAULT }


...
```

Now that the enum is defined, let's keep looking at the methods of an import plugin:

```
func _get_preset_count():
    return Presets.size()
```

The _get_preset_count() method returns the amount of presets that this plugins defines. We only have one preset now, but we can make this method future-proof by returning the size of our `Presets` enumeration.

```
func _get_preset_name(preset_index):
    match preset_index:
        Presets.DEFAULT:
            return "Default"
        _:
            return "Unknown"
```

Here we have the _get_preset_name() method, which gives names to the presets as they will be presented to the user, so be sure to use short and clear names.

We can use the `match` statement here to make the code more structured. This way it's easy to add new presets in the future. We use the catch all pattern to return something too. Although Godot won't ask for presets beyond the preset count you defined, it's always better to be on the safe side.

If you have only one preset you could simply return its name directly, but if you do this you have to be careful when you add more presets.

```
func _get_import_options(path, preset_index):
    match preset_index:
        Presets.DEFAULT:
            return [{
                       "name": "use_red_anyway",
                       "default_value": false
                    }]
        _:
            return []
```

This is the method which defines the available options. _get_import_options() returns an array of dictionaries, and each dictionary contains a few keys that are checked to customize the option as its shown to the user. The following table shows the possible keys:

Key	Type	Description
`name`	String	The name of the option. When showed, underscores become spaces and first letters are capitalized.
`default_value`	Any	The default value of the option for this preset.
`property_hint`	Enum value	One of the PropertyHint values to use as hint.

Key	Type	Description
`hint_string`	String	The hint text of the property. The same as you'd add in the `export` statement in GDScript.
`usage`	Enum value	One of the PropertyUsageFlags values to define the usage.

The `name` and `default_value` keys are **mandatory**, the rest are optional.

Note that the `_get_import_options` method receives the preset number, so you can configure the options for each different preset (especially the default value). In this example we use the `match` statement, but if you have lots of options and the presets only change the value you may want to create the array of options first and then change it based on the preset.

Warning

The `_get_import_options` method is called even if you don't define presets (by making `_get_preset_count` return zero). You have to return an array even it's empty, otherwise you can get errors.

```
func _get_option_visibility(path, option_name, options):
    return true
```

For the _get_option_visibility() method, we simply return `true` because all of our options (i.e. the single one we defined) are visible all the time.

If you need to make certain option visible only if another is set with a certain value, you can add the logic in this method.

The `import` method

The heavy part of the process, responsible for converting the files into resources, is covered by the _import() method. Our sample code is a bit long, so let's split in a few parts:

```
func _import(source_file, save_path, options,
r_platform_variants, r_gen_files):
    var file = FileAccess.open(source_file, FileAccess.READ)
    if file == null:
        return FileAccess.get_open_error()

    var line = file.get_line()
```

The first part of our import method opens and reads the source file. We use the FileAccess class to do that, passing the `source_file` parameter which is provided by the editor.

If there's an error when opening the file, we return it to let the editor know that the import wasn't successful.

```
var channels = line.split(",")
if channels.size() != 3:
    return ERR_PARSE_ERROR

var color
if options.use_red_anyway:
    color = Color8(255, 0, 0)
else:
    color = Color8(int(channels[0]), int(channels[1]),
int(channels[2]))
```

This code takes the line of the file it read before and splits it in pieces that are separated by a comma. If there are more or less than the three values, it considers the file invalid and reports an error.

Then it creates a new Color variable and sets its values according to the input file. If the `use_red_anyway` option is enabled, then it sets the color as a pure red instead.

```
var material = StandardMaterial3D.new()
material.albedo_color = color
```

This part makes a new StandardMaterial3D that is the imported resource. We create a new instance of it and then set its albedo color as the value we got before.

```
return ResourceSaver.save(material, "%s.%s" % [save_path,
_get_save_extension()])
```

This is the last part and quite an important one, because here we save the made resource to the disk. The path of the saved file is generated and informed by the editor via the `save_path` parameter. Note that this comes **without** the extension, so we add it using string formatting. For this we call the `_get_save_extension` method that we defined earlier, so we can be sure that they won't get out of sync.

We also return the result from the ResourceSaver.save() method, so if there's an error in this step, the editor will know about it.

Platform variants and generated files

You may have noticed that our plugin ignored two arguments of the `import` method. Those are *return arguments* (hence the `r` at the beginning of their name), which means that the editor will read from them after calling your import method. Both of them are arrays that you can fill with information.

The `r_platform_variants` argument is used if you need to import the resource differently depending on the target platform. While it's called *platform* variants, it is based on the presence of feature tags, so even the same platform can have multiple variants depending on the setup.

To import a platform variant, you need to save it with the feature tag before the extension, and then push the tag to the `r_platform_variants` array so the editor can know that you did.

For example, let's say we save a different material for a mobile platform. We would need to do something like the following:

```
r_platform_variants.push_back("mobile")
return ResourceSaver.save(mobile_material, "%s.%s.%s" %
[save_path, "mobile", _get_save_extension()])
```

The `r_gen_files` argument is meant for extra files that are generated during your import process and need to be kept. The editor will look at it to understand the dependencies and make sure the extra file is not inadvertently deleted.

This is also an array and should be filled with full paths of the files you save. As an example, let's create another material for the next pass and save it in a different file:

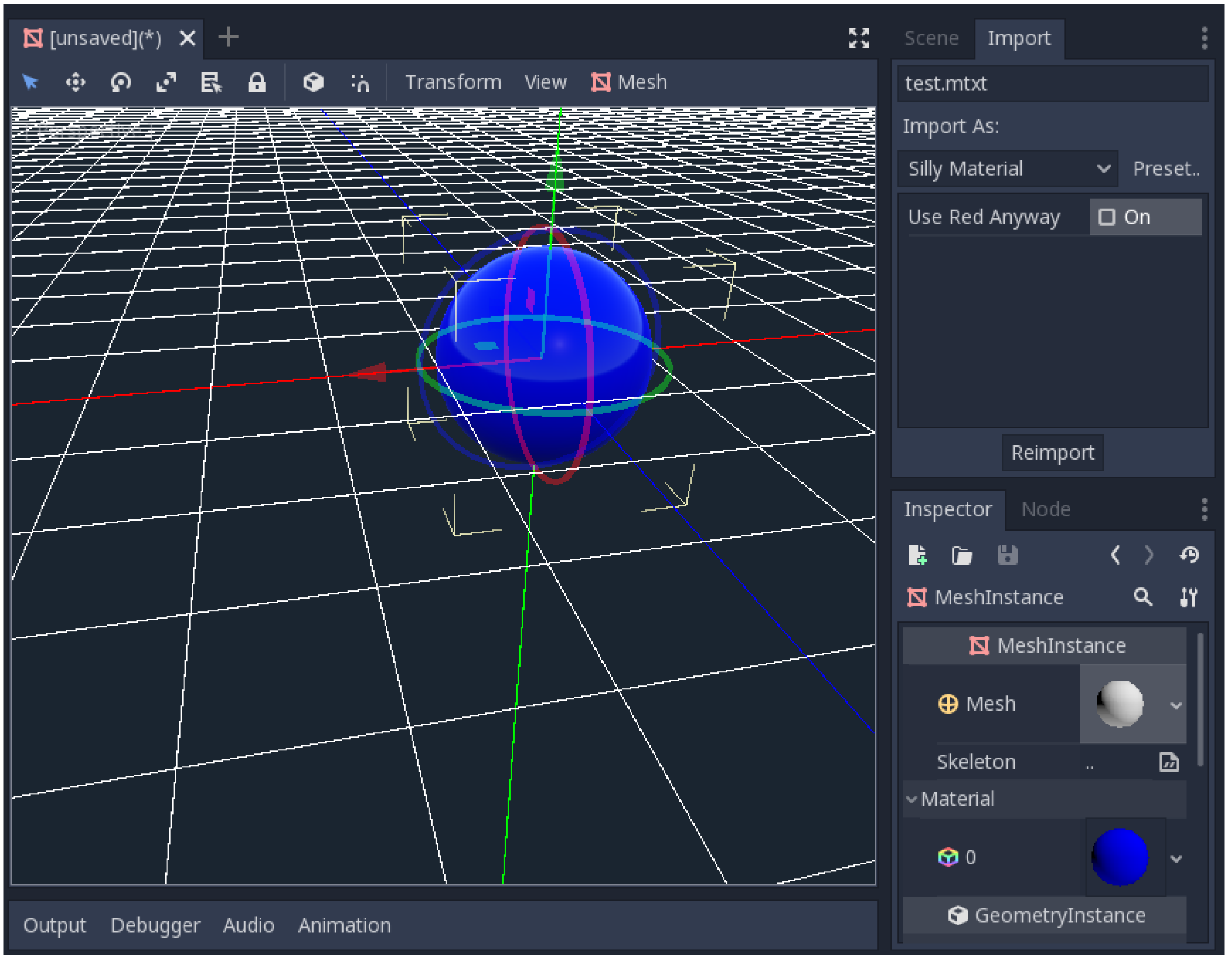
```
var next_pass = StandardMaterial3D.new()
next_pass.albedo_color = color.inverted()
var next_pass_path = "%s.next_pass.%s" % [save_path,
_get_save_extension()]

err = ResourceSaver.save(next_pass, next_pass_path)
if err != OK:
    return err
r_gen_files.push_back(next_pass_path)
```

Trying the plugin

This has been theoretical, but now that the import plugin is done, let's test it. Make sure you created the sample file (with the contents described in the introduction section) and save it as `test.mtxt`. Then activate the plugin in the Project Settings.

If everything goes well, the import plugin is added to the editor and the file system is scanned, making the custom resource appear on the FileSystem dock. If you select it and focus the Import dock, you can see the only option to select there.

Create a MeshInstance3D node in the scene, and for its Mesh property set up a new SphereMesh. Unfold the Material section in the Inspector and then drag the file from the FileSystem dock to the material property. The object will update in the viewport with the blue color of the imported material.


[unsaved](*)
Transform
View
Mesh
Scene
Import
Reimport
Inspector
Node
MeshInstance
MeshInstance
Mesh
Skeleton
Material
0
GeometryInstance
Output
Debugger
Audio
Animation

test.mtxt

Import As:

Silly Material

Preset..

Use Red Anyway ☐ On

Go to Import dock, enable the "Use Red Anyway" option, and click on "Reimport". This will update the imported material and should automatically update the view showing the red color instead.

And that's it! Your first import plugin is done! Now get creative and make plugins for your own beloved formats. This can be quite useful to write your data in a custom format and then use it in Godot as if they were native resources. This shows how the import system is powerful and extendable.

3D gizmo plugins

Introduction

3D gizmo plugins are used by the editor and custom plugins to define the gizmos attached to any kind of Node3D node.

This tutorial shows the two main approaches to defining your own custom gizmos. The first option works well for simple gizmos and creates less clutter in your plugin structure, and the second one will let you store some per-gizmo data.

Note

This tutorial assumes you already know how to make generic plugins. If in doubt, refer to the Making plugins page.

The EditorNode3DGizmoPlugin

Regardless of the approach we choose, we will need to create a new EditorNode3DGizmoPlugin. This will allow us to set a name for the new gizmo type and define other behaviors such as whether the gizmo can be hidden or not.

This would be a basic setup:

```
# my_custom_gizmo_plugin.gd
extends EditorNode3DGizmoPlugin
```

```
func _get_gizmo_name():
    return "CustomNode"
```

```
# MyCustomEditorPlugin.gd
@tool
extends EditorPlugin


const MyCustomGizmoPlugin = preload("res://addons/my-addon/my_custom_gizmo_plugin.gd")

var gizmo_plugin = MyCustomGizmoPlugin.new()


func _enter_tree():
    add_node_3d_gizmo_plugin(gizmo_plugin)


func _exit_tree():
    remove_node_3d_gizmo_plugin(gizmo_plugin)
```

For simple gizmos, inheriting EditorNode3DGizmoPlugin is enough. If you want to store some per-gizmo data or you are porting a Godot 3.0 gizmo to 3.1+, you should go with the second approach.

Simple approach

The first step is to, in our custom gizmo plugin, override the _has_gizmo() method so that it returns `true` when the node parameter is of our target type.

```
# ...


func _has_gizmo(node):
    return node is MyCustomNode3D
```

```
# ...
```

Then we can override methods like _redraw() or all the handle related ones.

```
# ...


func _init():
    create_material("main", Color(1, 0, 0))
    create_handle_material("handles")


func _redraw(gizmo):
    gizmo.clear()

    var node3d = gizmo.get_node_3d()

    var lines = PackedVector3Array()

    lines.push_back(Vector3(0, 1, 0))
    lines.push_back(Vector3(0, node3d.my_custom_value, 0))

    var handles = PackedVector3Array()

    handles.push_back(Vector3(0, 1, 0))
    handles.push_back(Vector3(0, node3d.my_custom_value, 0))

    gizmo.add_lines(lines, get_material("main", gizmo), false)
    gizmo.add_handles(handles, get_material("handles", gizmo),
[])


# ...
```

Note that we created a material in the _*init* method, and retrieved it in the _*redraw* method using get_material(). This method retrieves one of the material's variants depending on the state of the gizmo (selected and/or editable).

So the final plugin would look somewhat like this:

```
extends EditorNode3DGizmoPlugin


const MyCustomNode3D = preload("res://addons/my-
addon/my_custom_node_3d.gd")


func _init():
    create_material("main", Color(1,0,0))
    create_handle_material("handles")


func _has_gizmo(node):
    return node is MyCustomNode3D


func _redraw(gizmo):
    gizmo.clear()

    var node3d = gizmo.get_node_3d()

    var lines = PackedVector3Array()

    lines.push_back(Vector3(0, 1, 0))
    lines.push_back(Vector3(0, node3d.my_custom_value, 0))

    var handles = PackedVector3Array()

    handles.push_back(Vector3(0, 1, 0))
    handles.push_back(Vector3(0, node3d.my_custom_value, 0))

    gizmo.add_lines(lines, get_material("main", gizmo), false)
    gizmo.add_handles(handles, get_material("handles", gizmo),
[])


# You should implement the rest of handle-related callbacks
# (_get_handle_name(), _get_handle_value(), _commit_handle(),
...).
```

Note that we just added some handles in the *_redraw* method, but we still need to implement the rest of handle-related callbacks in EditorNode3DGizmoPlugin to get properly working handles.

Alternative approach

In some cases we want to provide our own implementation of EditorNode3DGizmo, maybe because we want to have some state stored in each gizmo or because we are porting an old gizmo plugin and we don't want to go through the rewriting process.

In these cases all we need to do is, in our new gizmo plugin, override _create_gizmo(), so it returns our custom gizmo implementation for the Node3D nodes we want to target.

```
# my_custom_gizmo_plugin.gd
extends EditorNode3DGizmoPlugin


const MyCustomNode3D = preload("res://addons/my-addon/my_custom_node_3d.gd")
const MyCustomGizmo = preload("res://addons/my-addon/my_custom_gizmo.gd")


func _init():
    create_material("main", Color(1, 0, 0))
    create_handle_material("handles")


func _create_gizmo(node):
    if node is MyCustomNode3D:
        return MyCustomGizmo.new()
    else:
        return null
```

This way all the gizmo logic and drawing methods can be implemented in a new class extending EditorNode3DGizmo, like so:

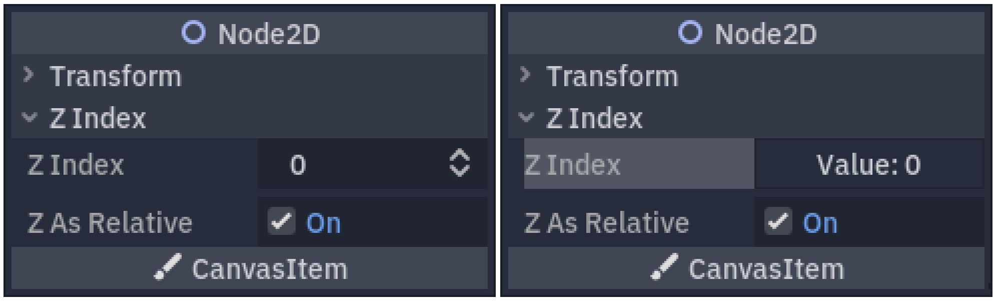
```
# my_custom_gizmo.gd
extends EditorNode3DGizmo


# You can store data in the gizmo itself (more useful when working with handles).
var gizmo_size = 3.0


func _redraw():
    clear()

    var node3d = get_node_3d()

    var lines = PackedVector3Array()

    lines.push_back(Vector3(0, 1, 0))
    lines.push_back(Vector3(gizmo_size, node3d.my_custom_value, 0))

    var handles = PackedVector3Array()

    handles.push_back(Vector3(0, 1, 0))
    handles.push_back(Vector3(gizmo_size, node3d.my_custom_value, 0))

    var material = get_plugin().get_material("main", self)
    add_lines(lines, material, false)

    var handles_material = get_plugin().get_material("handles", self)
    add_handles(handles, handles_material, [])


# You should implement the rest of handle-related callbacks
# (_get_handle_name(), _get_handle_value(), _commit_handle(), ...).
```

Note that we just added some handles in the *_redraw* method, but we still need to implement the rest of handle-related callbacks in EditorNode3DGizmo to get properly working handles.

Inspector plugins

The inspector dock allows you to create custom widgets to edit properties through plugins. This can be beneficial when working with custom datatypes and resources, although you can use the feature to change the inspector widgets for built-in types. You can design custom controls for specific properties, entire objects, and even separate controls associated with particular datatypes.

This guide explains how to use the EditorInspectorPlugin and EditorProperty classes to create a custom interface for integers, replacing the default behavior with a button that generates random values between 0 and 99.



Z Index 0
Z As Relative ☑ On

Z Index Value: 0
Z As Relative ☑ On

The default behavior on the left and the end result on the right.

Setting up your plugin

Create a new empty plugin to get started.

See also

See Making plugins guide to set up your new plugin.

Let's assume you've called your plugin folder `my_inspector_plugin`. If so, you should end up with a new `addons/my_inspector_plugin` folder that contains two files: `plugin.cfg` and `plugin.gd`.

As before, `plugin.gd` is a script extending EditorPlugin and you need to introduce new code for its `_enter_tree` and `_exit_tree` methods. To set up your inspector plugin, you must load its script, then create and add the instance by calling `add_inspector_plugin()`. If the plugin is disabled, you should remove the instance you have added by calling `remove_inspector_plugin()`.

Note

Here, you are loading a script and not a packed scene. Therefore you should use `new()` instead of `instantiate()`.

GDScript

```
# plugin.gd
@tool
extends EditorPlugin

var plugin


func _enter_tree():
    plugin = 
preload("res://addons/my_inspector_plugin/my_inspector_plugin.g
d").new()
    add_inspector_plugin(plugin)


func _exit_tree():
    remove_inspector_plugin(plugin)
```

C#

```
// Plugin.cs
#if TOOLS
using Godot;

[Tool]
public partial class Plugin : EditorPlugin
{
    private MyInspectorPlugin _plugin;

    public override void _EnterTree()
    {
        _plugin = new MyInspectorPlugin();
        AddInspectorPlugin(_plugin);
    }

    public override void _ExitTree()
    {
        RemoveInspectorPlugin(_plugin);
    }
}
#endif
```

Interacting with the inspector

To interact with the inspector dock, your `my_inspector_plugin.gd` script must extend the EditorInspectorPlugin class. This class provides several virtual methods that affect how the inspector handles properties.

To have any effect at all, the script must implement the `_can_handle()` method. This function is called for each edited Object and must return `true` if this plugin should handle the object or its properties.

Note

This includes any Resource attached to the object.

You can implement four other methods to add controls to the inspector at specific positions. The `_parse_begin()` and `_parse_end()` methods are called only once at the beginning and the end of parsing for each object, respectively. They can add controls at the top or bottom of the inspector layout by calling `add_custom_control()`.

As the editor parses the object, it calls the `_parse_category()` and `_parse_property()` methods. There, in addition to `add_custom_control()`, you can call both `add_property_editor()` and `add_property_editor_for_multiple_properties()`. Use these last two methods to specifically add EditorProperty-based controls.

GDScript

```
# my_inspector_plugin.gd
extends EditorInspectorPlugin

var RandomIntEditor =
preload("res://addons/my_inspector_plugin/random_int_editor.gd"
)


func _can_handle(object):
    # We support all objects in this example.
    return true


func _parse_property(object, type, name, hint_type,
hint_string, usage_flags, wide):
    # We handle properties of type integer.
    if type == TYPE_INT:
        # Create an instance of the custom property editor and
register
        # it to a specific property path.
        add_property_editor(name, RandomIntEditor.new())
        # Inform the editor to remove the default property
editor for
        # this property type.
        return true
    else:
        return false
```

C#

```
// MyInspectorPlugin.cs
#if TOOLS
using Godot;

public partial class MyInspectorPlugin : EditorInspectorPlugin
{
    public override bool _CanHandle(GodotObject @object)
    {
        // We support all objects in this example.
        return true;
    }

    public override bool _ParseProperty(GodotObject @object, Variant.Type type,
        string name, PropertyHint hintType, string hintString,
        PropertyUsageFlags usageFlags, bool wide)
    {
        // We handle properties of type integer.
        if (type == Variant.Type.Int)
        {
            // Create an instance of the custom property editor and register
            // it to a specific property path.
            AddPropertyEditor(name, new RandomIntEditor());
            // Inform the editor to remove the default property editor for
            // this property type.
            return true;
        }

        return false;
    }
}
#endif
```

Adding an interface to edit properties

The EditorProperty class is a special type of Control that can interact with the inspector dock's edited objects. It doesn't display anything but can house any other control nodes, including complex scenes.

There are three essential parts to the script extending EditorProperty:

1. You must define the `_init()` method to set up the control nodes' structure.
2. You should implement the `_update_property()` to handle changes to the data from the outside.
3. A signal must be emitted at some point to inform the inspector that the control has changed the property using `emit_changed`.

You can display your custom widget in two ways. Use just the default `add_child()` method to display it to the right of the property name, and use `add_child()` followed by `set_bottom_editor()` to position it below the name.

GDScript

```
# random_int_editor.gd
extends EditorProperty


# The main control for editing the property.
var property_control = Button.new()
# An internal value of the property.
var current_value = 0
# A guard against internal changes when the property is
updated.
var updating = false
```

```
func _init():
    # Add the control as a direct child of EditorProperty node.
    add_child(property_control)
    # Make sure the control is able to retain the focus.
    add_focusable(property_control)
    # Setup the initial state and connect to the signal to
track changes.
    refresh_control_text()
    property_control.pressed.connect(_on_button_pressed)


func _on_button_pressed():
    # Ignore the signal if the property is currently being
updated.
    if (updating):
        return

    # Generate a new random integer between 0 and 99.
    current_value = randi() % 100
    refresh_control_text()
    emit_changed(get_edited_property(), current_value)


func _update_property():
    # Read the current value from the property.
    var new_value = get_edited_object()[get_edited_property()]
    if (new_value == current_value):
        return

    # Update the control with the new value.
    updating = true
    current_value = new_value
    refresh_control_text()
    updating = false

func refresh_control_text():
    property_control.text = "Value: " + str(current_value)
```

C#

```
// RandomIntEditor.cs
#if TOOLS
```

```csharp
using Godot;

public partial class RandomIntEditor : EditorProperty
{
    // The main control for editing the property.
    private Button _propertyControl = new Button();
    // An internal value of the property.
    private int _currentValue = 0;
    // A guard against internal changes when the property is updated.
    private bool _updating = false;

    public RandomIntEditor()
    {
        // Add the control as a direct child of EditorProperty node.
        AddChild(_propertyControl);
        // Make sure the control is able to retain the focus.
        AddFocusable(_propertyControl);
        // Setup the initial state and connect to the signal to track changes.
        RefreshControlText();
        _propertyControl.Pressed += OnButtonPressed;
    }

    private void OnButtonPressed()
    {
        // Ignore the signal if the property is currently being updated.
        if (_updating)
        {
            return;
        }

        // Generate a new random integer between 0 and 99.
        _currentValue = (int)GD.Randi() % 100;
        RefreshControlText();
        EmitChanged(GetEditedProperty(), _currentValue);
    }

    public override void _UpdateProperty()
    {
        // Read the current value from the property.
        var newValue =
(int)GetEditedObject().Get(GetEditedProperty());
        if (newValue == _currentValue)
        {
            return;
        }

        // Update the control with the new value.
        _updating = true;
        _currentValue = newValue;
        RefreshControlText();
        _updating = false;
    }

    private void RefreshControlText()
    {
        _propertyControl.Text = $"Value: {_currentValue}";
    }
}
#endif
```

Using the example code above you should be able to make a custom widget that replaces the default SpinBox control for integers with a Button that generates random values.

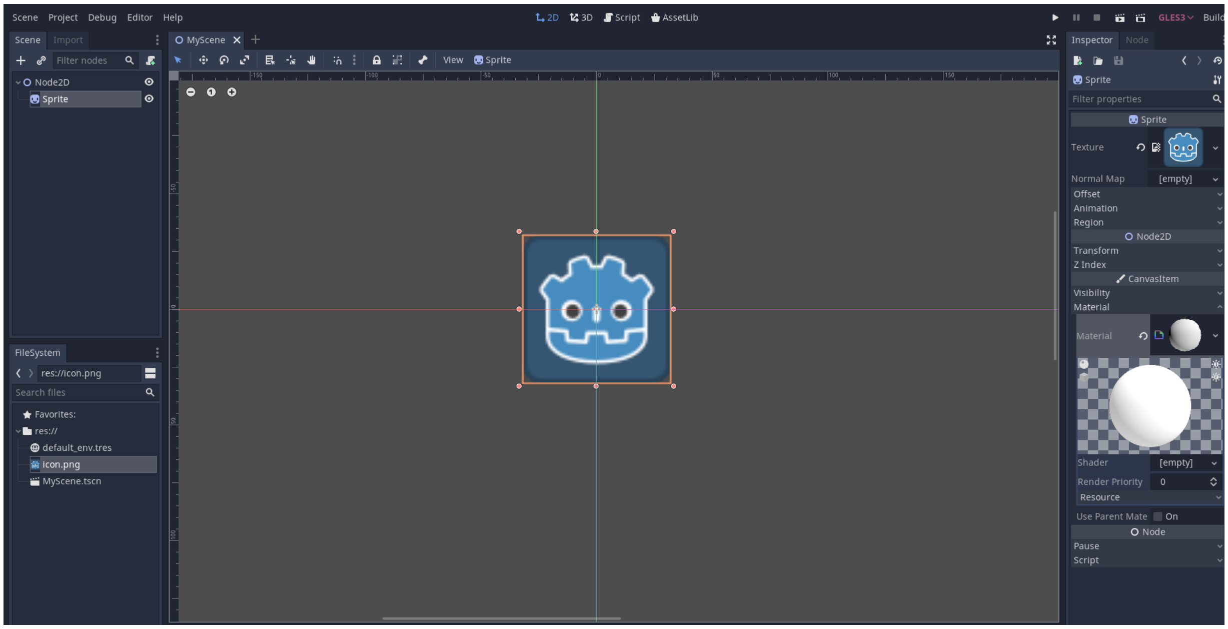
Visual Shader plugins

Visual Shader plugins are used to create custom VisualShader nodes in GDScript.

The creation process is different from usual editor plugins. You do not need to create a `plugin.cfg` file to register it; instead, create and save a script file and it will be ready to use, provided the custom node is registered with `class_name`.

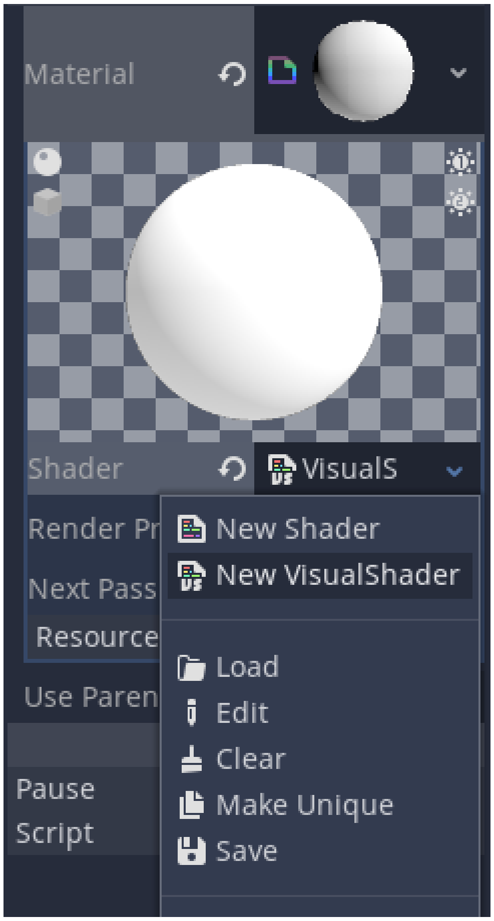
This short tutorial will explain how to make a Perlin-3D noise node (original code from this GPU noise shaders plugin [https://github.com/curly-brace/Godot-3.0-Noise-Shaders/blob/master/assets/gpu_noise_shaders/classic_perlin3d.tres].

Create a Sprite2D and assign a ShaderMaterial to its material slot:


Scene Project Debug Editor Help
2D 3D Script AssetLib
GLES3 Build
Scene Import
Filter nodes
Node2D
Sprite
MyScene
View Sprite
Inspector Node
Sprite
Filter properties
Texture
Offset
Animation
Region
Node2D
Transform
Z Index
CanvasItem
Visibility
Material
Resource
Node
Pause
Script
FileSystem
res://icon.png
Search files
Favorites:
res://
default_env.tres
icon.png
MyScene.tscn

Normal Map [empty]
Shader [empty]
Render Priority 0
Use Parent Mate On

Assign VisualShader to the shader slot of the material:

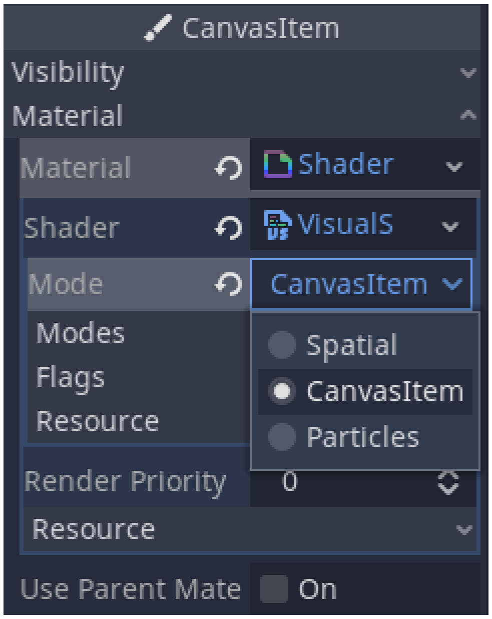


Material

Shader VisualS

Render Pr
Next Pass
Resource
Use Paren
Pause
Script
New Shader
New VisualShader
Load
Edit
Clear
Make Unique
Save

Don't forget to change its mode to "CanvasItem" (if you are using a Sprite2D):



Material Shader

Shader VisualS

Mode CanvasItem

Render Priority 0

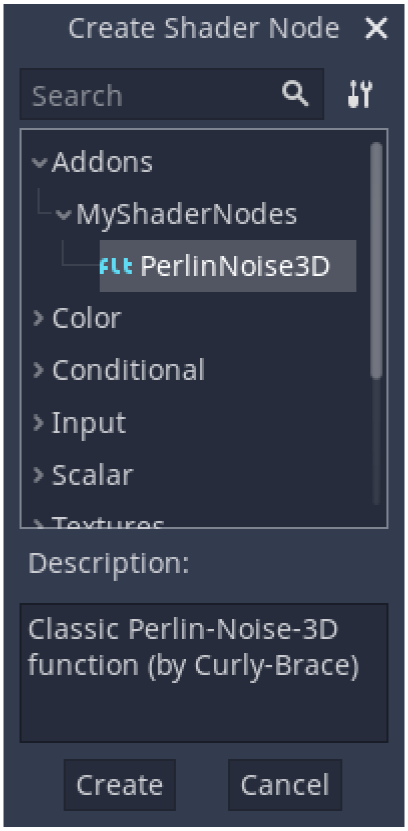
Use Parent Mate On

CanvasItem
Visibility
Material
Modes
Flags
Resource
Spatial
CanvasItem
Particles
Resource

Create a script which derives from VisualShaderNodeCustom. This is all you need to initialize your plugin.

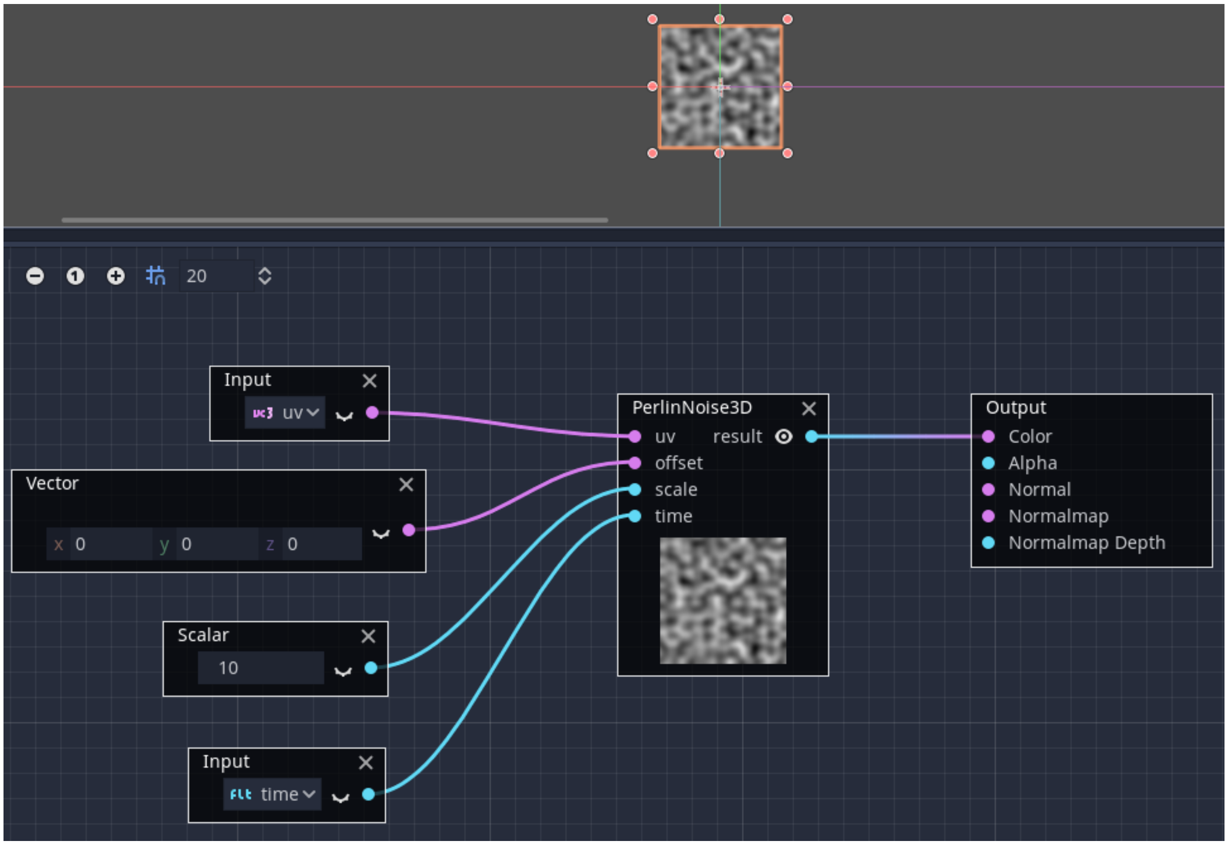
```
# perlin_noise_3d.gd
@tool
extends VisualShaderNodeCustom
class_name VisualShaderNodePerlinNoise3D


func _get_name():
    return "PerlinNoise3D"


func _get_category():
    return "MyShaderNodes"


func _get_description():
    return "Classic Perlin-Noise-3D function (by Curly-Brace)"


func _init():
    set_input_port_default_value(2, 0.0)


func _get_return_icon_type():
    return VisualShaderNode.PORT_TYPE_SCALAR


func _get_input_port_count():
    return 4


func _get_input_port_name(port):
    match port:
        0:
            return "uv"
        1:
            return "offset"
        2:
            return "scale"
        3:
            return "time"


func _get_input_port_type(port):
    match port:
        0:
            return VisualShaderNode.PORT_TYPE_VECTOR_3D
        1:
            return VisualShaderNode.PORT_TYPE_VECTOR_3D
        2:
            return VisualShaderNode.PORT_TYPE_SCALAR
        3:
            return VisualShaderNode.PORT_TYPE_SCALAR


func _get_output_port_count():
    return 1


func _get_output_port_name(port):
    return "result"


func _get_output_port_type(port):
    return VisualShaderNode.PORT_TYPE_SCALAR


func _get_global_code(mode):
    return """
        vec3 mod289_3(vec3 x) {
            return x - floor(x * (1.0 / 289.0)) * 289.0;
        }

        vec4 mod289_4(vec4 x) {
            return x - floor(x * (1.0 / 289.0)) * 289.0;
        }

        vec4 permute(vec4 x) {
            return mod289_4(((x * 34.0) + 1.0) * x);
        }

        vec4 taylorInvSqrt(vec4 r) {
            return 1.79284291400159 - 0.85373472095314 * r;
        }

        vec3 fade(vec3 t) {
            return t * t * t * (t * (t * 6.0 - 15.0) + 10.0);
        }

        // Classic Perlin noise.
        float cnoise(vec3 P) {
            vec3 Pi0 = floor(P); // Integer part for indexing.
            vec3 Pi1 = Pi0 + vec3(1.0); // Integer part + 1.
            Pi0 = mod289_3(Pi0);
            Pi1 = mod289_3(Pi1);
            vec3 Pf0 = fract(P); // Fractional part for
interpolation.
            vec3 Pf1 = Pf0 - vec3(1.0); // Fractional part -
1.0.
            vec4 ix = vec4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
            vec4 iy = vec4(Pi0.yy, Pi1.yy);
            vec4 iz0 = vec4(Pi0.z);
            vec4 iz1 = vec4(Pi1.z);

            vec4 ixy = permute(permute(ix) + iy);
            vec4 ixy0 = permute(ixy + iz0);
            vec4 ixy1 = permute(ixy + iz1);

            vec4 gx0 = ixy0 * (1.0 / 7.0);
            vec4 gy0 = fract(floor(gx0) * (1.0 / 7.0)) - 0.5;
            gx0 = fract(gx0);
            vec4 gz0 = vec4(0.5) - abs(gx0) - abs(gy0);
            vec4 sz0 = step(gz0, vec4(0.0));
            gx0 -= sz0 * (step(0.0, gx0) - 0.5);
            gy0 -= sz0 * (step(0.0, gy0) - 0.5);

            vec4 gx1 = ixy1 * (1.0 / 7.0);
            vec4 gy1 = fract(floor(gx1) * (1.0 / 7.0)) - 0.5;
            gx1 = fract(gx1);
            vec4 gz1 = vec4(0.5) - abs(gx1) - abs(gy1);
            vec4 sz1 = step(gz1, vec4(0.0));
            gx1 -= sz1 * (step(0.0, gx1) - 0.5);
            gy1 -= sz1 * (step(0.0, gy1) - 0.5);

            vec3 g000 = vec3(gx0.x, gy0.x, gz0.x);
            vec3 g100 = vec3(gx0.y, gy0.y, gz0.y);
            vec3 g010 = vec3(gx0.z, gy0.z, gz0.z);
            vec3 g110 = vec3(gx0.w, gy0.w, gz0.w);
            vec3 g001 = vec3(gx1.x, gy1.x, gz1.x);
            vec3 g101 = vec3(gx1.y, gy1.y, gz1.y);
            vec3 g011 = vec3(gx1.z, gy1.z, gz1.z);
            vec3 g111 = vec3(gx1.w, gy1.w, gz1.w);

            vec4 norm0 = taylorInvSqrt(vec4(dot(g000, g000),
dot(g010, g010), dot(g100, g100), dot(g110, g110)));
            g000 *= norm0.x;
            g010 *= norm0.y;
            g100 *= norm0.z;
            g110 *= norm0.w;
            vec4 norm1 = taylorInvSqrt(vec4(dot(g001, g001),
dot(g011, g011), dot(g101, g101), dot(g111, g111)));
            g001 *= norm1.x;
            g011 *= norm1.y;
            g101 *= norm1.z;
            g111 *= norm1.w;

            float n000 = dot(g000, Pf0);
            float n100 = dot(g100, vec3(Pf1.x, Pf0.yz));
            float n010 = dot(g010, vec3(Pf0.x, Pf1.y, Pf0.z));
            float n110 = dot(g110, vec3(Pf1.xy, Pf0.z));
            float n001 = dot(g001, vec3(Pf0.xy, Pf1.z));
            float n101 = dot(g101, vec3(Pf1.x, Pf0.y, Pf1.z));
            float n011 = dot(g011, vec3(Pf0.x, Pf1.yz));
            float n111 = dot(g111, Pf1);

            vec3 fade_xyz = fade(Pf0);
            vec4 n_z = mix(vec4(n000, n100, n010, n110),
vec4(n001, n101, n011, n111), fade_xyz.z);
            vec2 n_yz = mix(n_z.xy, n_z.zw, fade_xyz.y);
            float n_xyz = mix(n_yz.x, n_yz.y, fade_xyz.x);
            return 2.2 * n_xyz;
        }
    """


func _get_code(input_vars, output_vars, mode, type):
    return output_vars[0] + " = cnoise(vec3((%s.xy + %s.xy) *
%s, %s)) * 0.5 + 0.5;" % [input_vars[0], input_vars[1],
input_vars[2], input_vars[3]]
```

Save it and open the Visual Shader. You should see your new node type within the member's dialog under the Addons category (if you can't see your new node, try restarting the editor):


Create Shader Node
Search
Addons
MyShaderNodes
PerlinNoise3D
Color
Conditional
Input
Scalar
Textures
Description:
Create
Cancel

Classic Perlin-Noise-3D function (by Curly-Brace)

Place it on a graph and connect the required ports:


20
Input
uv
Vector
x 0
y 0
z 0
Scalar
10
Input
time
PerlinNoise3D
uv
result
offset
scale
time
Output
Color
Alpha
Normal
Normalmap
Normalmap Depth

That is everything you need to do, as you can see it is easy to create your own custom VisualShader nodes!

Running code in the editor

What is `@tool`?

`@tool` is a powerful line of code that, when added at the top of your script, makes it execute in the editor. You can also decide which parts of the script execute in the editor, which in game, and which in both.

You can use it for doing many things, but it is mostly useful in level design for visually presenting things that are hard to predict ourselves. Here are some use cases:

- If you have a cannon that shoots cannonballs affected by physics (gravity), you can draw the cannonball's trajectory in the editor, making level design a lot easier.
- If you have jumppads with varying jump heights, you can draw the maximum jump height a player would reach if it jumped on one, also making level design easier.
- If your player doesn't use a sprite, but draws itself using code, you can make that drawing code execute in the editor to see your player.

Danger

`@tool` scripts run inside the editor, and let you access the scene tree of the currently edited scene. This is a powerful feature which also comes with caveats, as the editor does not include protections for potential misuse of `@tool` scripts. Be **extremely** cautious when manipulating the scene tree, especially via Node.queue_free, as it can cause crashes if you free a node while the editor runs logic involving it.

How to use `@tool`

To turn a script into a tool, add the `@tool` annotation at the top of your code.

To check if you are currently in the editor, use: `Engine.is_editor_hint()`.

For example, if you want to execute some code only in the editor, use:

GDScript

```
if Engine.is_editor_hint():
    # Code to execute when in editor.
```

C#

```
if (Engine.IsEditorHint())
{
    // Code to execute when in editor.
}
```

On the other hand, if you want to execute code only in game, simply negate the same statement:

GDScript

```
if not Engine.is_editor_hint():
    # Code to execute when in game.
```

C#

```
if (!Engine.IsEditorHint())
{
    // Code to execute when in game.
}
```

Pieces of code that do not have either of the 2 conditions above will run both in-editor and in-game.

Here is how a `_process()` function might look for you:

GDScript

```
func _process(delta):
    if Engine.is_editor_hint():
        # Code to execute in editor.

    if not Engine.is_editor_hint():
        # Code to execute in game.

    # Code to execute both in editor and in game.
```

C#

```
public override void _Process(double delta)
{
    if (Engine.IsEditorHint())
    {
        // Code to execute in editor.
    }

    if (!Engine.IsEditorHint())
    {
        // Code to execute in game.
    }

    // Code to execute both in editor and in game.
}
```

Important information

Any other GDScript that your tool script uses must *also* be a tool. Any GDScript without `@tool` used by the editor will act like an empty file!

Extending a `@tool` script does not automatically make the extending script a `@tool`. Omitting `@tool` from the extending script will disable tool behavior from the super class. Therefore the extending script should also specify the `@tool` annotation.

Modifications in the editor are permanent. For example, in the next section when we remove the script, the node will keep its rotation. Be careful to avoid making unwanted modifications.

Try `@tool` out

Add a `Sprite2D` node to your scene and set the texture to Godot icon. Attach and open a script, and change it to this:

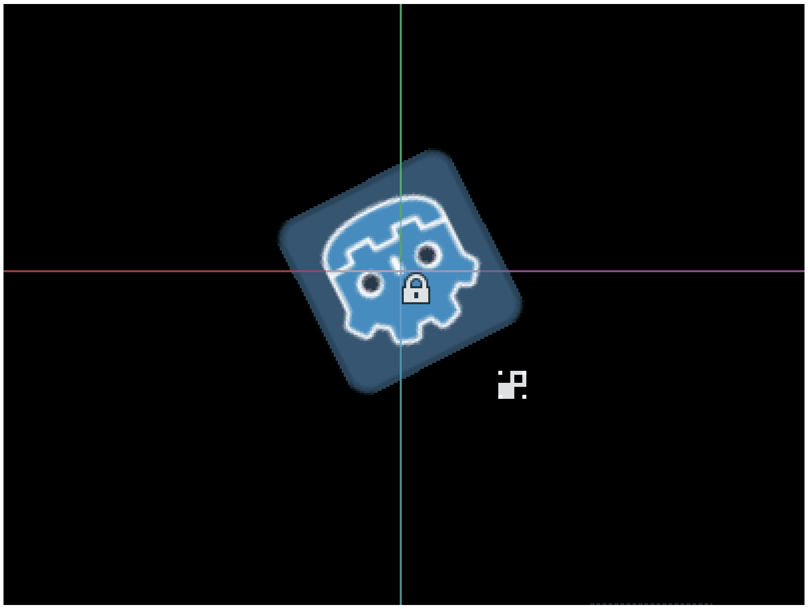
GDScript

```
@tool
extends Sprite2D

func _process(delta):
    rotation += PI * delta
```

C#

```
using Godot;

[Tool]
public partial class MySprite : Sprite2D
{
    public override void _Process(double delta)
    {
        Rotation += Mathf.Pi * (float)delta;
    }
}
```

Save the script and return to the editor. You should now see your object rotate. If you run the game, it will also rotate.



Note

If you don't see the changes, reload the scene (close it and open it again).

Now let's choose which code runs when. Modify your `_process()` function to look like this:

GDScript

```
func _process(delta):
    if Engine.is_editor_hint():
        rotation += PI * delta
    else:
        rotation -= PI * delta
```

C#

```
public override void _Process(double delta)
{
    if (Engine.IsEditorHint())
    {
        Rotation += Mathf.Pi * (float)delta;
    }
    else
    {
        Rotation -= Mathf.Pi * (float)delta;
    }
}
```

Save the script. Now the object will spin clockwise in the editor, but if you run the game, it will spin counter-clockwise.

Editing variables

Add and export a variable speed to the script. To update the speed and also reset the rotation angle add a setter `set(new_speed)` which is executed with the input from the inspector. Modify `_process()` to include the rotation speed.

GDScript

```
@tool
extends Sprite2D


@export var speed = 1:
    # Update speed and reset the rotation.
    set(new_speed):
        speed = new_speed
        rotation = 0


func _process(delta):
    rotation += PI * delta * speed
```

C#

```
using Godot;

[Tool]
public partial class MySprite : Sprite2D
{
    private float _speed = 1;

    [Export]
    public float Speed
    {
        get => _speed;
        set
        {
            // Update speed and reset the rotation.
            _speed = value;
            Rotation = 0;
        }
    }

    public override void _Process(double delta)
    {
        Rotation += Mathf.Pi * (float)delta * speed;
    }
}
```

Note

Code from other nodes doesn't run in the editor. Your access to other nodes is limited. You can access the tree and nodes, and their default properties, but you can't access user variables. If you want to do so, other nodes have to run in the editor too. Autoload nodes cannot be accessed in the editor at all.

Getting notified when resources change

Sometimes you want your tool to use a resource. However, when you change a property of that resource in the editor, the `set()` method of your tool will not be called.

GDScript

```
@tool
class_name MyTool
extends Node

@export var resource: MyResource:
    set(new_resource):
        resource = new_resource
        _on_resource_set()

# This will only be called when you create, delete, or paste a resource.
# You will not get an update when tweaking properties of it.
func _on_resource_set():
    print("My resource was set!")
```

C#

```
using Godot;

[Tool]
public partial class MyTool : Node
{
    private MyResource _resource;

    [Export]
    public MyResource Resource
    {
        get => _resource;
        set
        {
            _resource = value;
            OnResourceSet();
        }
    }
}
```

```
// This will only be called when you create, delete, or paste a
resource.
// You will not get an update when tweaking properties of it.
private void OnResourceSet()
{
    GD.Print("My resource was set!");
}
```

To get around this problem you first have to make your resource a tool and make it emit the `changed` signal whenever a property is set:

GDScript

```
# Make Your Resource a tool.
@tool
class_name MyResource
extends Resource

@export var property = 1:
    set(new_setting):
        property = new_setting
        # Emit a signal when the property is changed.
        changed.emit()
```

C#

```
using Godot;

[Tool]
public partial class MyResource : Resource
{
    private float _property = 1;

    [Export]
    public float Property
    {
        get => _property;
        set
        {
            _property = value;
            // Emit a signal when the property is changed.
            EmitChanged();
        }
    }
}
```

You then want to connect the signal when a new resource is set:

GDScript

```
@tool
class_name MyTool
extends Node

@export var resource: MyResource:
    set(new_resource):
        resource = new_resource
        # Connect the changed signal as soon as a new resource
is being added.
        resource.changed.connect(_on_resource_changed)

func _on_resource_changed():
    print("My resource just changed!")
```

C#

```
using Godot;

[Tool]
public partial class MyTool : Node
{
    private MyResource _resource;

    [Export]
    public MyResource Resource
    {
        get => _resource;
        set
        {
            _resource = value;
            // Connect the changed signal as soon as a new
resource is being added.
```

```
            _resource.Changed += OnResourceChanged;
        }
    }
}

private void OnResourceChanged()
{
    GD.Print("My resource just changed!");
}
```

Lastly, remember to disconnect the signal as the old resource being used and changed somewhere else would cause unneeded updates.

GDScript

```
@export var resource: MyResource:
    set(new_resource):
        # Disconnect the signal if the previous resource was
not null.
        if resource != null:
            resource.changed.disconnect(_on_resource_changed)
        resource = new_resource
        resource.changed.connect(_on_resource_changed)
```

C#

```
[Export]
public MyResource Resource
{
    get => _resource;
    set
    {
        // Disconnect the signal if the previous resource was
not null.
        if (_resource != null)
        {
            _resource.Changed -= OnResourceChanged;
        }
        _resource = value;
        _resource.Changed += OnResourceChanged;
    }
}
```

Reporting node configuration warnings

Godot uses a *node configuration warning* system to warn users about incorrectly configured nodes. When a node isn't configured correctly, a yellow warning sign appears next to the node's name in the Scene dock. When you hover or click on the icon, a warning message pops up. You can use this feature in your scripts to help you and your team avoid mistakes when setting up scenes.

When using node configuration warnings, when any value that should affect or remove the warning changes, you need to call update_configuration_warnings . By default, the warning only updates when closing and reopening the scene.

GDScript

```
# Use setters to update the configuration warning
automatically.
@export var title = "":
    set(p_title):
        if p_title != title:
            title = p_title
            update_configuration_warnings()

@export var description = "":
    set(p_description):
        if p_description != description:
            description = p_description
            update_configuration_warnings()


func _get_configuration_warnings():
    var warnings = []
    if title == "":
        warnings.append("Please set `title` to a non-empty value.")

    if description.length() >= 100:
        warnings.append("`description` should be less than 100 characters long.")

    # Returning an empty array means "no warning".
    return warnings
```

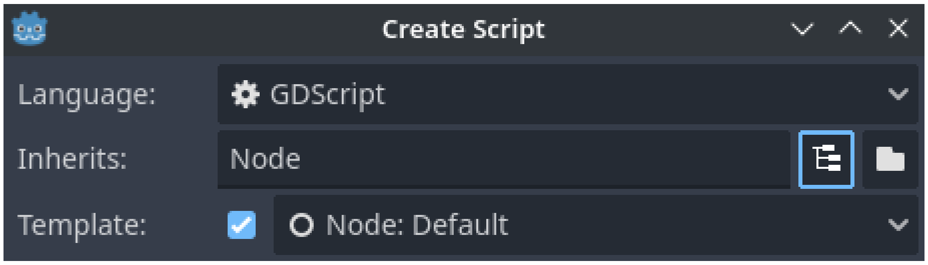
Running one-off scripts using EditorScript

Sometimes, you need to run code just one time to automate a certain task that is not available in the editor out of the box. Some examples might be:

- Use as a playground for GDScript or C# scripting without having to run a project. `print()` output is displayed in the editor Output panel.
- Scale all light nodes in the currently edited scene, as you noticed your level ends up looking too dark or too bright after placing lights where desired.
- Replace nodes that were copy-pasted with scene instances to make them easier to modify later.

This is available in Godot by extending EditorScript in a script. This provides a way to run individual scripts in the editor without having to create an editor plugin.

To create an EditorScript, right-click a folder or empty space in the FileSystem dock then choose **New > Script...**. In the script creation dialog, click the tree icon to choose an object to extend from (or enter `EditorScript` directly in the field on the left, though note this is case-sensitive):



Language: GDScript

Inherits: Node

Template: ☑ Node: Default

Creating an editor script in the script editor creation dialog

This will automatically select a script template that is suited for EditorScripts, with a `_run()` method already inserted:

```
@tool
extends EditorScript

# Called when the script is executed (using File -> Run in
Script Editor).
func _run():
    pass
```

This `_run()` method is executed when you use **File > Run** or the keyboard shortcut `Ctrl + Shift + X` while the EditorScript is the currently open script in the script editor. This keyboard shortcut is only effective when currently focused on the script editor.

Scripts that extend EditorScript must be `@tool` scripts to function.

Warning

EditorScripts have no undo/redo functionality, so **make sure to save your scene before running one** if the script is designed to modify any data.

To access nodes in the currently edited scene, use the EditorScript.get_scene method which returns the root Node of the currently edited scene. Here's an example that recursively gets all nodes in the currently edited scene and doubles the range of all OmniLight3D nodes:

```
@tool
extends EditorScript

func _run():
    for node in get_all_children(get_scene()):
        if node is OmniLight3D:
            # Don't operate on instanced subscene children, as
changes are lost
            # when reloading the scene.
            # See the "Instancing scenes" section below for a
description of `owner`.
            var is_instanced_subscene_child = node !=
get_scene() and node.owner != get_scene()
            if not is_instanced_subscene_child:
                node.omni_range *= 2.0

# This function is recursive: it calls itself to get lower
levels of child nodes as needed.
# `children_acc` is the accumulator parameter that allows this
function to work.
# It should be left to its default value when you call this
function directly.
func get_all_children(in_node, children_acc = []):
    children_acc.push_back(in_node)
    for child in in_node.get_children():
        children_acc = get_all_children(child, children_acc)

    return children_acc
```

Tip

You can change the currently edited scene at the top of the editor even while the Script view is open. This will affect the return value of EditorScript.get_scene, so make sure you've selected the scene you intend to iterate upon before running the script.

Instancing scenes

You can instantiate packed scenes normally and add them to the scene currently opened in the editor. By default, nodes or scenes added with Node.add_child(node) are **not** visible in the Scene tree dock and are **not** persisted to disk. If you wish the node or scene to be visible in the scene tree dock and persisted to disk when saving the scene, you need to set the child node's owner property to the currently edited scene root.

If you are using `@tool`:

GDScript

```
func _ready():
    var node = Node3D.new()
    add_child(node) # Parent could be any node in the scene

    # The line below is required to make the node visible in
the Scene tree dock
    # and persist changes made by the tool script to the saved
scene file.
    node.owner = get_tree().edited_scene_root
```

C#

```
public override void _Ready()
{
    var node = new Node3D();
    AddChild(node); // Parent could be any node in the scene

    // The line below is required to make the node visible in
the Scene tree dock
    // and persist changes made by the tool script to the saved
scene file.
    node.Owner = GetTree().EditedSceneRoot;
}
```

If you are using EditorScript:

GDScript

```
func _run():
    # `parent` could be any node in the scene.
    var parent = get_scene().get_node("Parent")
    var node = Node3D.new()
    parent.add_child(node)

    # The line below is required to make the node visible in
the Scene tree dock
    # and persist changes made by the tool script to the saved
scene file.
    node.owner = get_scene()
```

C#

```
public override void _Run()
{
    // `parent` could be any node in the scene.
    var parent = GetScene().GetNode("Parent");
    var node = new Node3D();
    parent.AddChild(node);

    // The line below is required to make the node visible in
the Scene tree dock
    // and persist changes made by the tool script to the saved
scene file.
    node.Owner = GetScene();
}
```

Warning

Using `@tool` improperly can yield many errors. It is advised to first write the code how you want it, and only then add the `@tool` annotation to the top. Also, make sure to separate code that runs in-editor from code that runs in-game. This way, you can find bugs more easily.

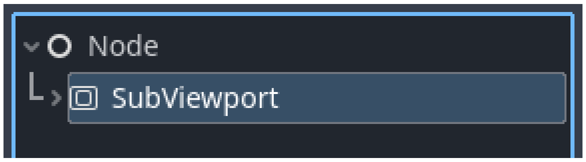
Rendering

- Using Viewports
- Multiple resolutions
- Fixing jitter, stutter and input lag
- The Compositor

Using Viewports

Introduction

Think of a Viewport as a screen onto which the game is projected. In order to see the game, we need to have a surface on which to draw it. That surface is the Root Viewport.



SubViewports are a kind of Viewport that can be added to the scene so that there are multiple surfaces to draw on. When we are drawing to a SubViewport, we call it a render target. We can access the contents of a render target by accessing its corresponding texture. By using a SubViewport as render target, we can either render multiple scenes simultaneously or we can render to a ViewportTexture which is applied to an object in the scene, for example a dynamic skybox.

SubViewports have a variety of use cases, including:

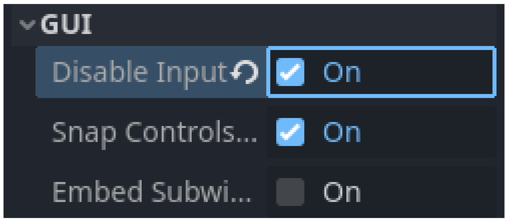
- Rendering 3D objects within a 2D game
- Rendering 2D elements in a 3D game
- Rendering dynamic textures
- Generating procedural textures at runtime
- Rendering multiple cameras in the same scene

What all these use cases have in common is that you are given the ability to draw objects to a texture as if it were another screen and can then choose what to do with the resulting texture.

Another kind of Viewports in Godot are Windows. They allow their content to be projected onto a window. While the Root Viewport is a Window, they are less flexible. If you want to use the texture of a Viewport, you'll be working with SubViewports most of the time.

Input

Viewports are also responsible for delivering properly adjusted and scaled input events to their children nodes. By default SubViewports don't automatically receive input, unless they receive it from their direct SubViewportContainer parent node. In this case, input can be disabled with the Disable Input property.



GUI

Disable Input ☑ On

Snap Controls... ☑ On

Embed Subwi... ☐ On

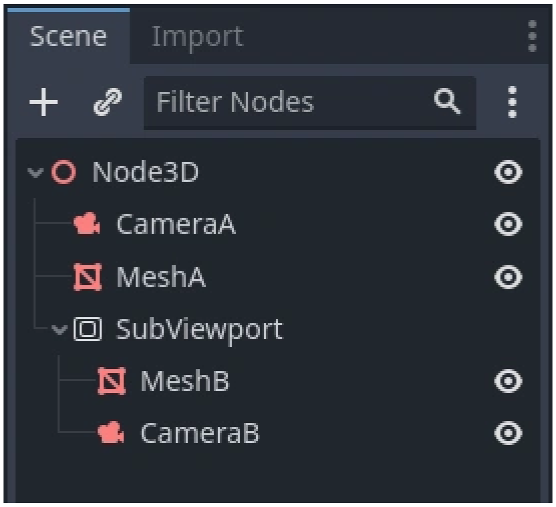
For more information on how Godot handles input, please read the Input Event Tutorial.

Listener

Godot supports 3D sound (in both 2D and 3D nodes). More on this can be found in the Audio Streams Tutorial. For this type of sound to be audible, the Viewport needs to be enabled as a listener (for 2D or 3D). If you are using a SubViewport to display your World3D or World2D, don't forget to enable this!

Cameras (2D & 3D)

When using a Camera3D or Camera2D, it will always display on the closest parent Viewport (going towards the root). For example, in the following hierarchy:



`CameraA` will display on the Root Viewport and it will draw `MeshA`. `CameraB` will be captured by the SubViewport along with `MeshB`. Even though `MeshB` is in the scene hierarchy, it will still not be drawn to the Root Viewport. Similarly, `MeshA` will not be visible from the SubViewport because SubViewports only capture nodes below them in the hierarchy.

There can only be one active camera per Viewport, so if there is more than one, make sure that the desired one has the current property set, or make it the current camera by calling:

GDScript

```
camera.make_current()
```

C#

```
camera.MakeCurrent();
```

By default, cameras will render all objects in their world. In 3D, cameras can use their cull_mask property combined with the

VisualInstance3D's layer property to restrict which objects are rendered.

Scale & stretching

SubViewports have a size property, which represents the size of the SubViewport in pixels. For SubViewports which are children of SubViewportContainers, these values are overridden, but for all others, this sets their resolution.

It is also possible to scale the 2D content and make the SubViewport resolution different from the one specified in size, by calling:

GDScript

```
sub_viewport.set_size_2d_override(Vector2i(width, height)) # Custom size for 2D.
sub_viewport.set_size_2d_override_stretch(true) # Enable stretch for custom size.
```

C#

```
subViewport.Size2DOverride = new Vector2I(width, height); // Custom size for 2D.
subViewport.Size2DOverrideStretch = true; // Enable stretch for custom size.
```

For information on scaling and stretching with the Root Viewport visit the Multiple Resolutions Tutorial

Worlds

For 3D, a Viewport will contain a World3D. This is basically the universe that links physics and rendering together. Node3D-based nodes will register using the World3D of the closest Viewport. By default, newly created Viewports do not contain a World3D but use the same as their parent Viewport. The Root Viewport always contains a World3D, which is the one objects are rendered to by default.

A World3D can be set in a Viewport using the World 3D property, that will separate all children nodes of this Viewport and will prevent them from interacting with the parent Viewport's World3D. This is especially useful in scenarios where, for example, you might want to show a separate character in 3D imposed over the game (like in StarCraft).

As a helper for situations where you want to create Viewports that display single objects and don't want to create a World3D, Viewport has the option to use its Own World3D. This is useful when you want to instance 3D characters or objects in World2D.

For 2D, each Viewport always contains its own World2D. This suffices in most cases, but in case sharing them may be desired, it is possible to do so by setting world_2d on the Viewport through code.

For an example of how this works, see the demo projects 3D in 2D [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/3d_in_2d] and 2D in 3D [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/2d_in_3d] respectively.

Capture

It is possible to query a capture of the Viewport contents. For the Root Viewport, this is effectively a screen capture. This is done with the following code:

GDScript

```
# Retrieve the captured Image using get_image().
var img = get_viewport().get_texture().get_image()
```

```
# Convert Image to ImageTexture.
var tex = ImageTexture.create_from_image(img)
# Set sprite texture.
sprite.texture = tex
```

C#

```
// Retrieve the captured Image using get_image().
var img = GetViewport().GetTexture().GetImage();
// Convert Image to ImageTexture.
var tex = ImageTexture.CreateFromImage(img);
// Set sprite texture.
sprite.Texture = tex;
```

But if you use this in `_ready()` or from the first frame of the Viewport's initialization, you will get an empty texture because there is nothing to get as texture. You can deal with it using (for example):

GDScript

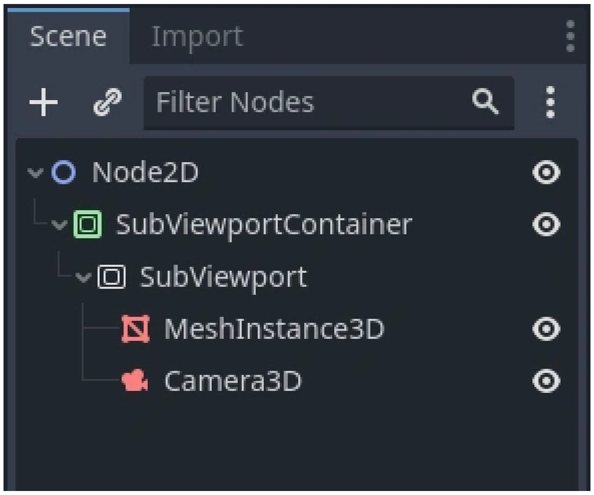
```
# Wait until the frame has finished before getting the texture.
await RenderingServer.frame_post_draw
# You can get the image after this.
```

C#

```
// Wait until the frame has finished before getting the
texture.
await
RenderingServer.Singleton.ToSignal(RenderingServer.SignalName.F
ramePostDraw);
// You can get the image after this.
```

Viewport Container

If the SubViewport is a child of a SubViewportContainer, it will become active and display anything it has inside. The layout looks like this:


Scene
Import
Filter Nodes
Node2D
SubViewportContainer
SubViewport
MeshInstance3D
Camera3D

The SubViewport will cover the area of its parent SubViewportContainer completely if Stretch is set to `true` in the SubViewportContainer.

Note

The size of the SubViewportContainer cannot be smaller than the size of the SubViewport.

Rendering

Due to the fact that the Viewport is an entryway into another rendering surface, it exposes a few rendering properties that can be different from the project settings. You can choose to use a different level of MSAA for each Viewport. The default behavior is `Disabled`.

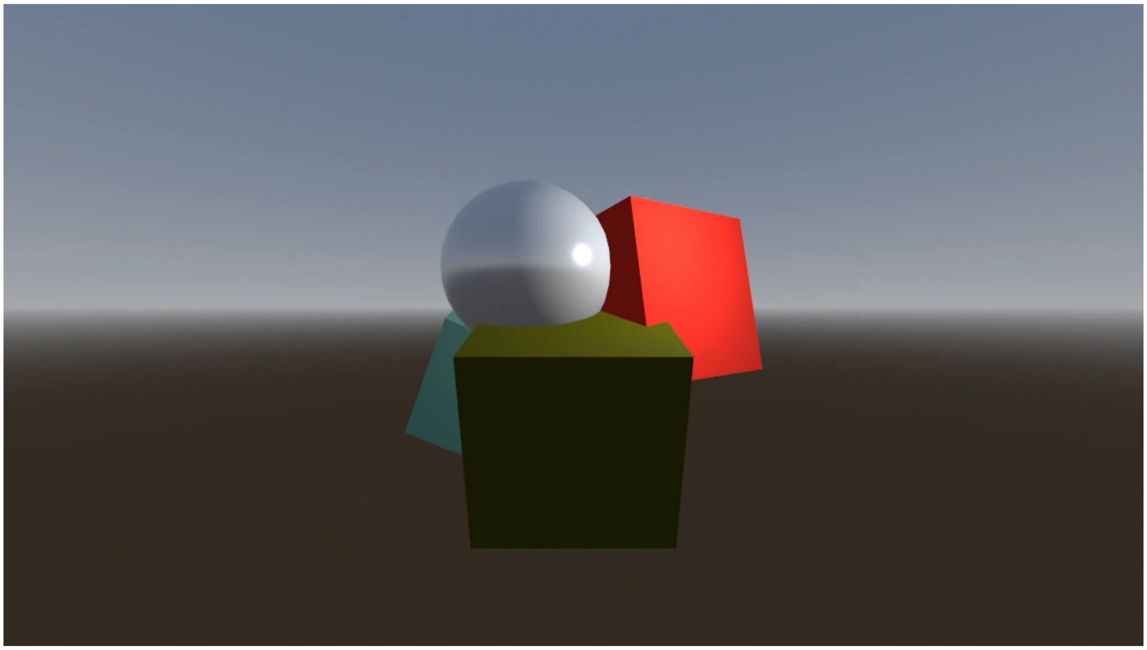
If you know that the Viewport is only going to be used for 2D, you can Disable 3D. Godot will then restrict how the Viewport is drawn. Disabling 3D is slightly faster and uses less memory compared to enabled 3D. It's a good idea to disable 3D if your viewport doesn't render anything in 3D.

Note

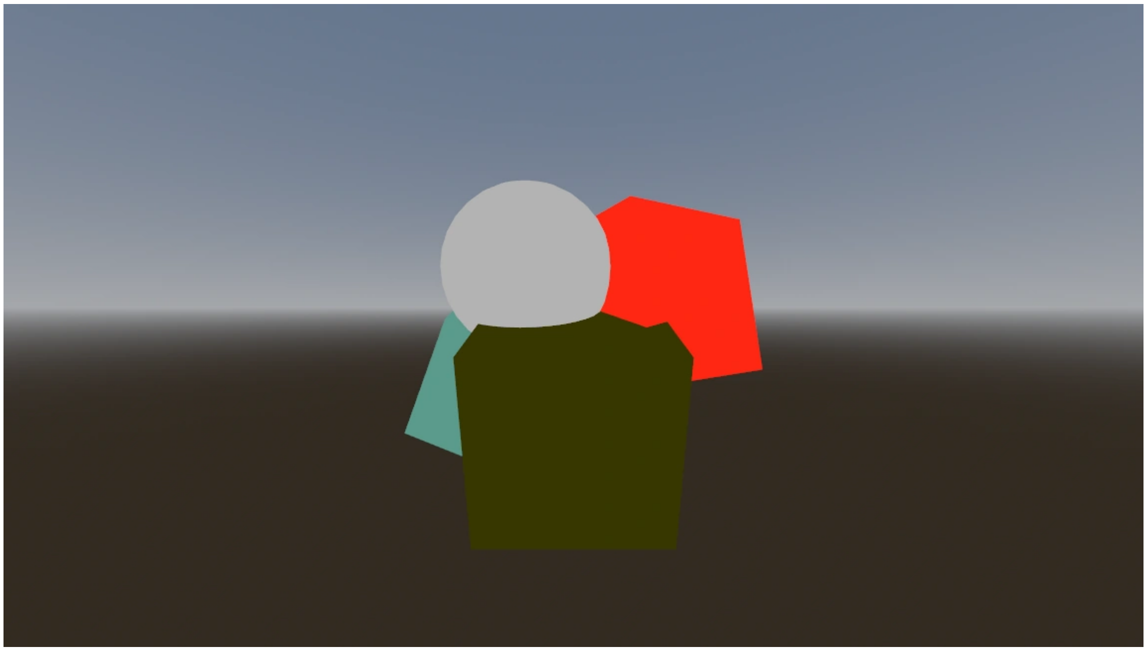
If you need to render 3D shadows in the viewport, make sure to set the viewport's positional_shadow_atlas_size property to a value higher than `0`. Otherwise, shadows won't be rendered. By default, the equivalent project setting is set to `4096` on desktop platforms and `2048` on mobile platforms.

Godot also provides a way of customizing how everything is drawn inside Viewports using Debug Draw. Debug Draw allows you to specify a mode which determines how the Viewport will display things drawn inside it. Debug Draw is `Disabled` by default. Some other options are `Unshaded`, `Overdraw`, and `Wireframe`. For a full list, refer to the Viewport Documentation.

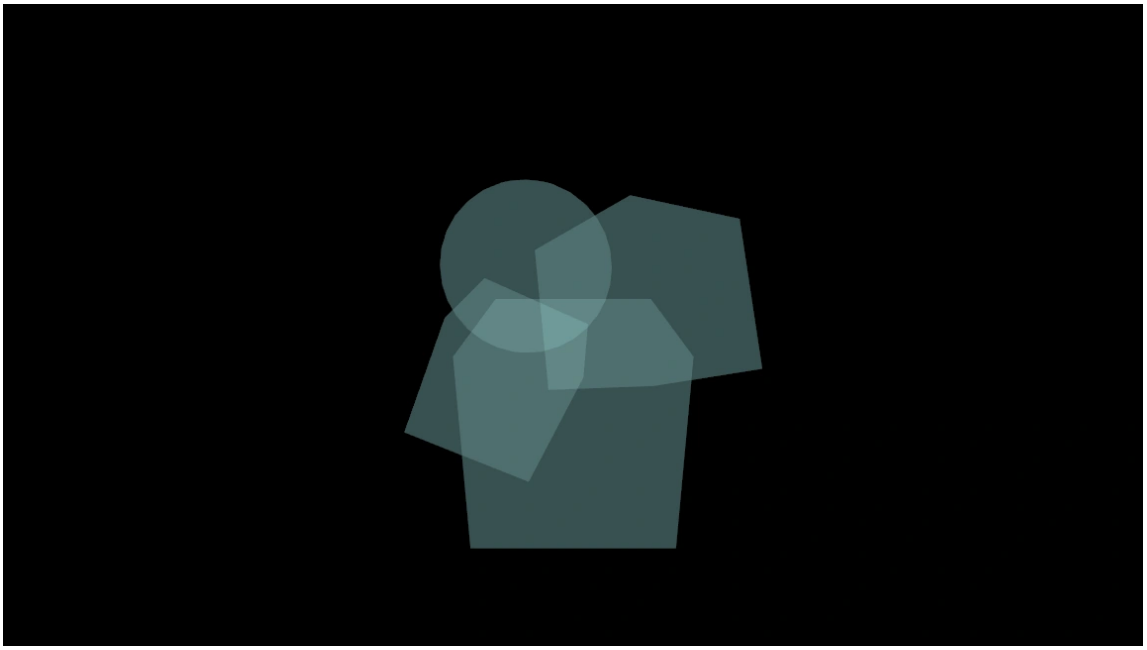
- **Debug Draw = Disabled** (default): The scene is drawn normally.



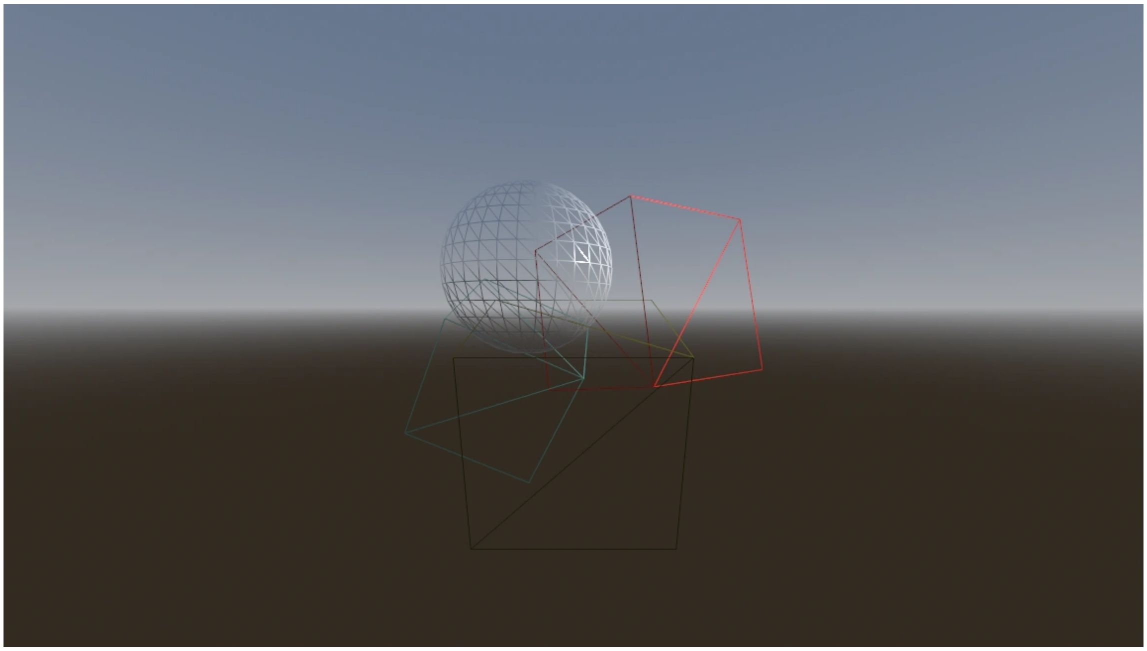
- **Debug Draw = Unshaded**: Unshaded draws the scene without using lighting information so all the objects appear flatly colored in their albedo color.



- **Debug Draw = Overdraw**: Overdraw draws the meshes semi-transparent with an additive blend so you can see how the meshes overlap.



- **Debug Draw = Wireframe**: Wireframe draws the scene using only the edges of triangles in the meshes.



Note

Debug Draw modes are currently **not** supported when using the Compatibility rendering method. They will appear as regular draw modes.

Render target

When rendering to a SubViewport, whatever is inside will not be visible in the scene editor. To display the contents, you have to draw the SubViewport's ViewportTexture somewhere. This can be requested via code using (for example):

GDScript

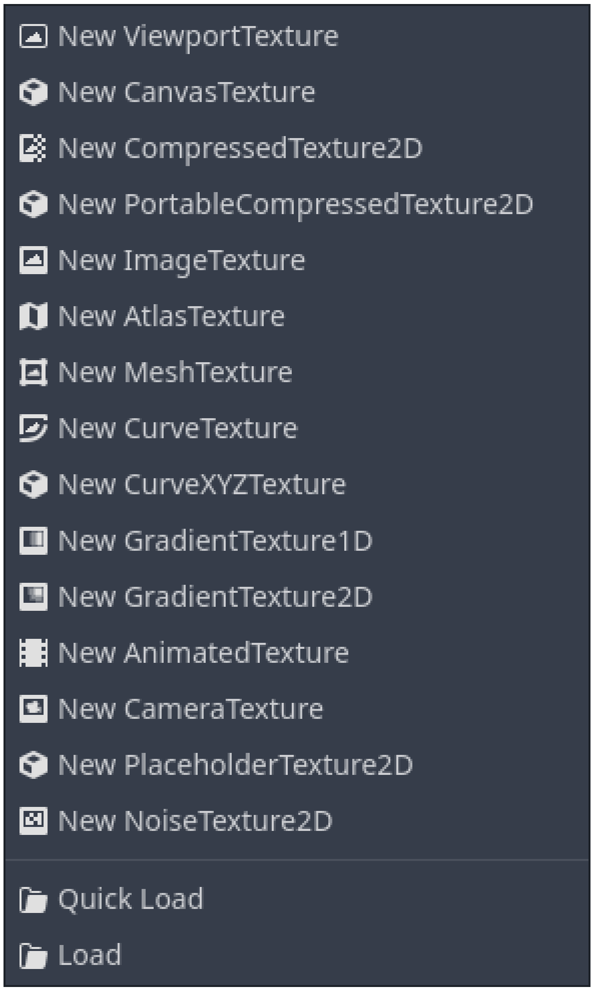
```
# This gives us the ViewportTexture.
var tex = viewport.get_texture()
```

```
sprite.texture = tex
```

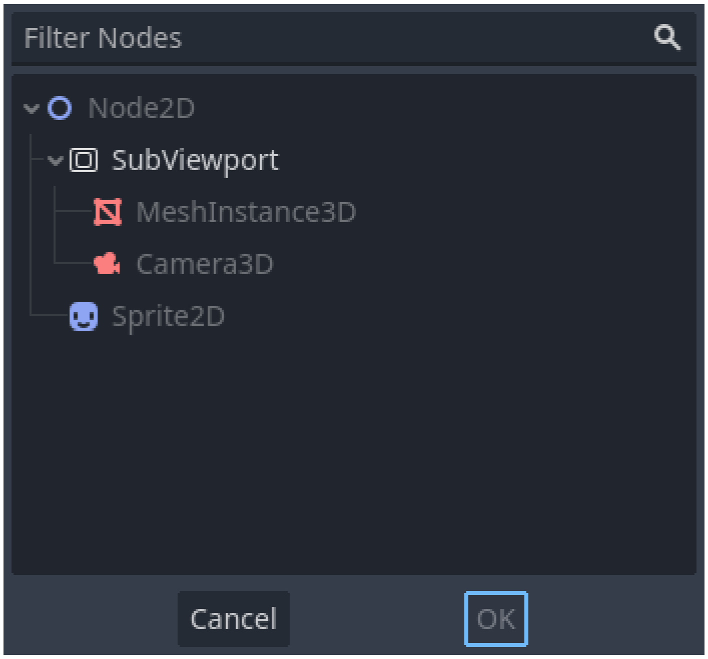
C#

```
// This gives us the ViewportTexture.
var tex = viewport.GetTexture();
sprite.Texture = tex;
```

Or it can be assigned in the editor by selecting "New ViewportTexture"


New ViewportTexture
New CanvasTexture
New CompressedTexture2D
New PortableCompressedTexture2D
New ImageTexture
New AtlasTexture
New MeshTexture
New CurveTexture
New CurveXYZTexture
New GradientTexture1D
New GradientTexture2D
New AnimatedTexture
New CameraTexture
New PlaceholderTexture2D
New NoiseTexture2D
Quick Load
Load

and then selecting the Viewport you want to use.



Every frame, the Viewport's texture is cleared away with the default clear color (or a transparent color if Transparent BG is set to `true`). This can be changed by setting Clear Mode to `Never` or `Next Frame`. As the name implies, Never means the texture will never be cleared, while next frame will clear the texture on the next frame and then set itself to Never.

By default, re-rendering of the SubViewport happens when its ViewportTexture has been drawn in a frame. If visible, it will be rendered, otherwise, it will not. This behavior can be changed by setting Update Mode to `Never`, `Once`, `Always`, or `When Parent Visible`. Never and Always will never or always re-render respectively. Once will re-render the next frame and change to Never afterwards. This can be used to manually update the Viewport. This flexibility allows users to render an image once and then use the texture without incurring the cost of rendering every frame.

Note

Make sure to check the Viewport demos. They are available in the viewport folder of the demos archive, or at https://github.com/godotengine/godot-demo-projects/tree/master/viewport.

Multiple resolutions

The problem of multiple resolutions

Developers often have trouble understanding how to best support multiple resolutions in their games. For desktop and console games, this is more or less straightforward, as most screen aspect ratios are 16:9 and resolutions are standard (720p, 1080p, 1440p, 4K, …).

For mobile games, at first, it was easy. For many years, the iPhone and iPad used the same resolution. When *Retina* was implemented, they just doubled the pixel density; most developers had to supply assets in default and double resolutions.

Nowadays, this is no longer the case, as there are plenty of different screen sizes, densities, and aspect ratios. Non-conventional sizes are also becoming increasingly popular, such as ultrawide displays.

For 3D games, there is not much of a need to support multiple resolutions (from the aesthetic point of view). The 3D geometry will just fill the screen based on the field of view, disregarding the aspect ratio. The main reason one may want to support this, in this case, is for *performance* reasons (running in lower resolution to increase frames per second).

For 2D and game UIs, this is a different matter, as art needs to be created using specific pixel sizes in software such as Photoshop, GIMP or Krita.

Since layouts, aspect ratios, resolutions, and pixel densities can change so much, it is no longer possible to design UIs for every specific screen. Another method must be used.

One size fits all

The most common approach is to use a single *base* resolution and then fit it to everything else. This resolution is how most players are expected to play the game (given their hardware). For mobile, Google has useful stats [https://developer.android.com/about/dashboards] online, and for desktop, Steam also does [https://store.steampowered.com/hwsurvey/].

As an example, Steam shows that the most common *primary display resolution* is 1920×1080, so a sensible approach is to develop a game for this resolution, then handle scaling for different sizes and aspect ratios.

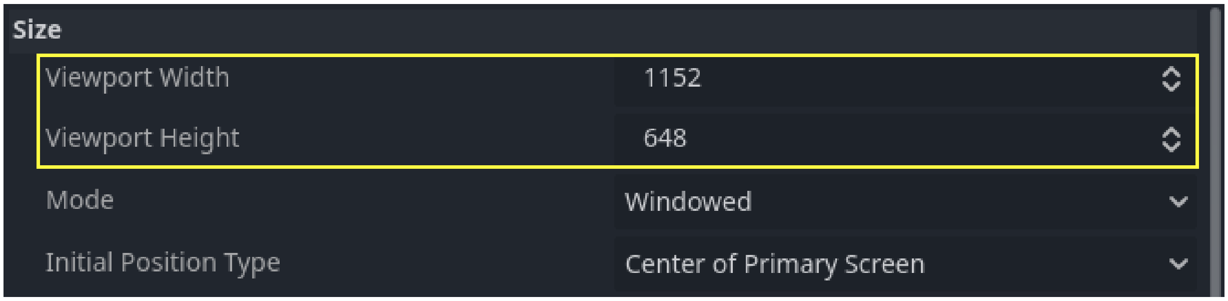
Godot provides several useful tools to do this easily.

See also

You can see how Godot's support for multiple resolutions works in action using the Multiple Resolutions and Aspect Ratios demo project [https://github.com/godotengine/godot-demo-projects/tree/master/gui/multiple_resolutions].

Base size

A base size for the window can be specified in the Project Settings under **Display** → **Window**.



Size

Viewport Width 1152

Viewport Height 648

Mode Windowed

Initial Position Type Center of Primary Screen

However, what it does is not completely obvious; the engine will *not* attempt to switch the monitor to this resolution. Rather, think of this setting as the "design size", i.e. the size of the area that you work with in the editor. This setting corresponds directly to the size of the blue rectangle in the 2D editor.

There is often a need to support devices with screen and window sizes that are different from this base size. Godot offers many ways to control how the viewport will be resized and stretched to different screen sizes.

Note

On this page, *window* refers to the screen area allotted to your game by the system, while *viewport* refers to the root object (accessible from `get_tree().root`) which the game controls to fill this screen area. This viewport is a Window instance. Recall from the introduction that *all* Window objects are viewports.

To configure the stretch base size at runtime from a script, use the `get_tree().root.content_scale_size` property (see Window.content_scale_size). Changing this value can indirectly change the size of 2D elements. However, to provide a user-accessible scaling option, using Stretch Scale is recommended as it's easier to adjust.

Note

Godot follows a modern approach to multiple resolutions. The engine will never change the monitor's resolution on its own. While changing the monitor's resolution is the most efficient approach, it's also the least reliable approach as it can leave the monitor stuck on a low resolution if the game crashes. This is especially common on macOS or Linux which don't handle resolution changes as well as Windows.

Changing the monitor's resolution also removes any control from the game developer over filtering and aspect ratio stretching, which can be important to ensure correct display for pixel art games.

On top of that, changing the monitor's resolution makes alt-tabbing in and out of a game much slower since the monitor has to change resolutions every time this is done.

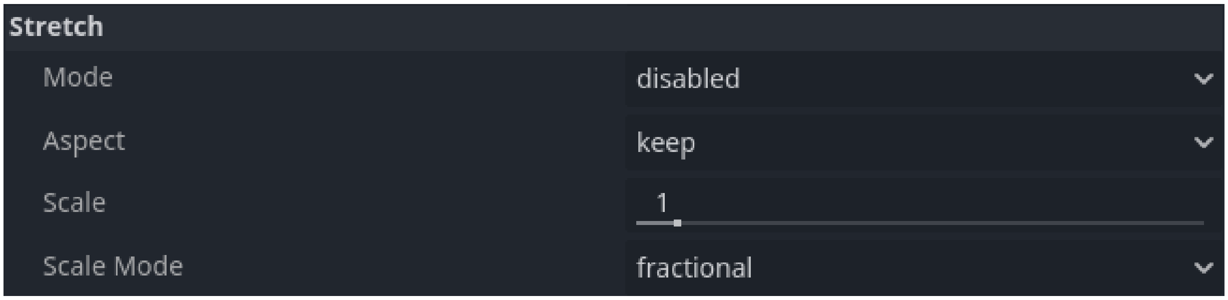
Resizing

There are several types of devices, with several types of screens, which in turn have different pixel density and resolutions. Handling all of them can be a lot of work, so Godot tries to make the developer's life a little easier. The Viewport node has several functions to handle resizing, and the root node of the scene tree is always a viewport (scenes loaded are instanced as a child of it, and it can always be accessed by calling `get_tree().root` or `get_node("/root")`).

In any case, while changing the root Viewport params is probably the most flexible way to deal with the problem, it can be a lot of work, code and guessing, so Godot provides a set of parameters in the project settings to handle multiple resolutions.

Stretch settings

Stretch settings are located in the project settings and provide several options:



Stretch

Mode: disabled

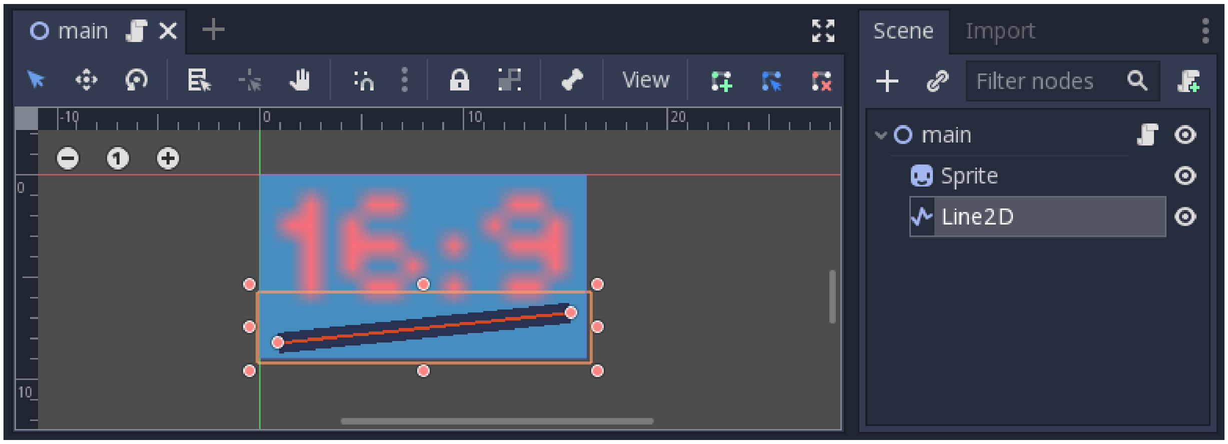
Aspect: keep

Scale: 1

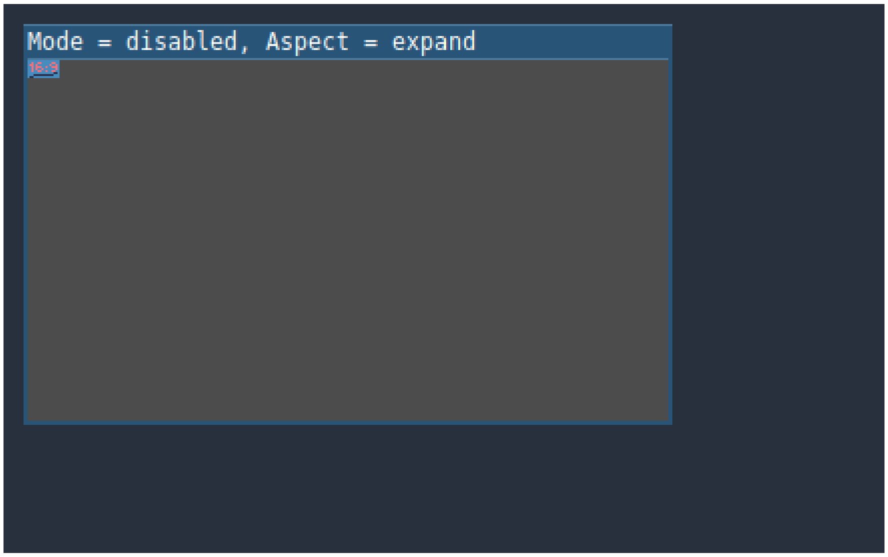
Scale Mode: fractional

Stretch Mode

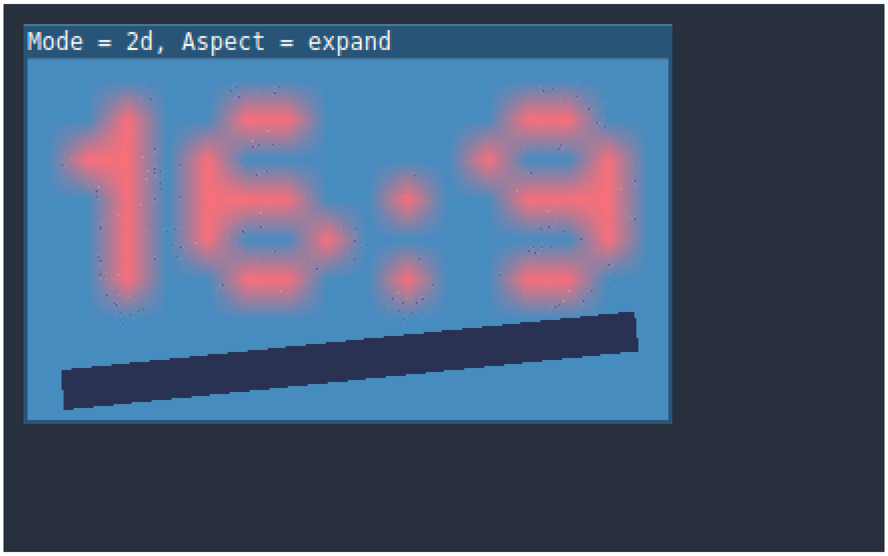
The **Stretch Mode** setting defines how the base size is stretched to fit the resolution of the window or screen. The animations below use a "base size" of just 16×9 pixels to demonstrate the effect of different stretch modes. A single sprite, also 16×9 pixels in size, covers the entire viewport, and a diagonal Line2D is added on top of it:



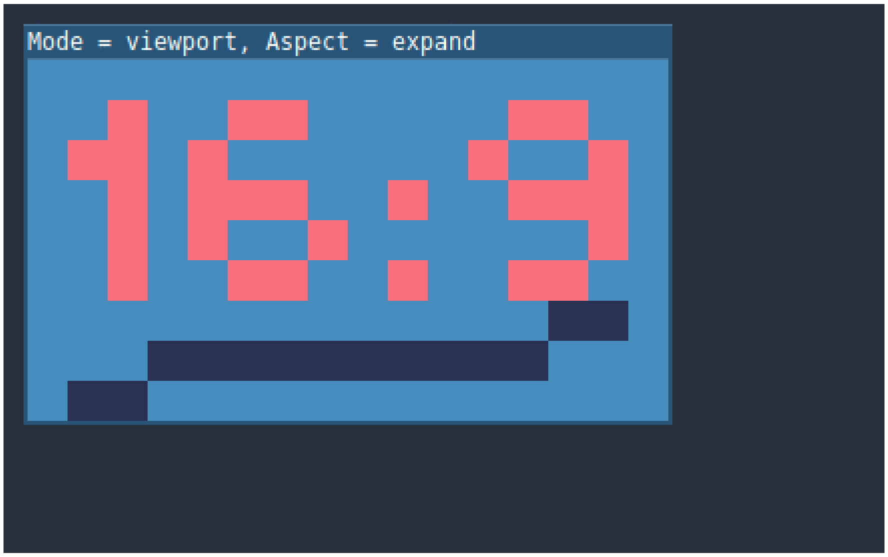
- **Stretch Mode = Disabled** (default): No stretching happens. One unit in the scene corresponds to one pixel on the screen. In this mode, the **Stretch Aspect** setting has no effect.


Mode = disabled, Aspect = expand

- **Stretch Mode = Canvas Items**: In this mode, the base size specified in width and height in the project settings is stretched to cover the whole screen (taking the **Stretch Aspect** setting into account). This means that everything is rendered directly at the target resolution. 3D is unaffected, while in 2D, there is no longer a 1:1 correspondence between sprite pixels and screen pixels, which may result in scaling artifacts.


Mode = 2d, Aspect = expand

- **Stretch Mode = Viewport**: Viewport scaling means that the size of the root Viewport is set precisely to the base size specified in the Project Settings' **Display** section. The scene is rendered to this viewport first. Finally, this viewport is scaled to fit the screen (taking the **Stretch Aspect** setting into account).



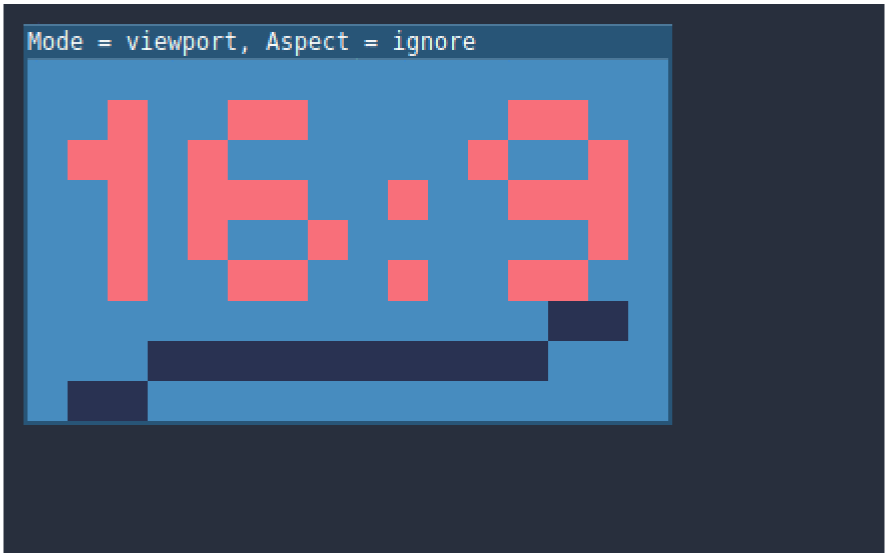
To configure the stretch mode at runtime from a script, use the `get_tree().root.content_scale_mode` property (see Window.content_scale_mode and the ContentScaleMode enum).

Stretch Aspect

The second setting is the stretch aspect. Note that this only takes effect if **Stretch Mode** is set to something other than **Disabled**.

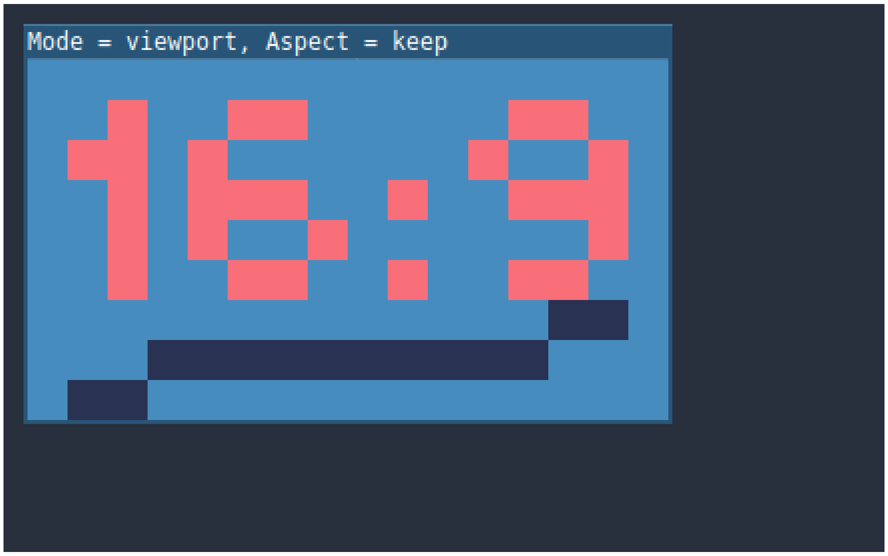
In the animations below, you will notice gray and black areas. The black areas are added by the engine and cannot be drawn into. The gray areas are part of your scene, and can be drawn to. The gray areas correspond to the region outside the blue frame you see in the 2D editor.

- **Stretch Aspect = Ignore**: Ignore the aspect ratio when stretching the screen. This means that the original resolution will be stretched to exactly fill the screen, even if it's wider or narrower. This may result in nonuniform stretching: things looking wider or taller than designed.


Mode = viewport, Aspect = ignore

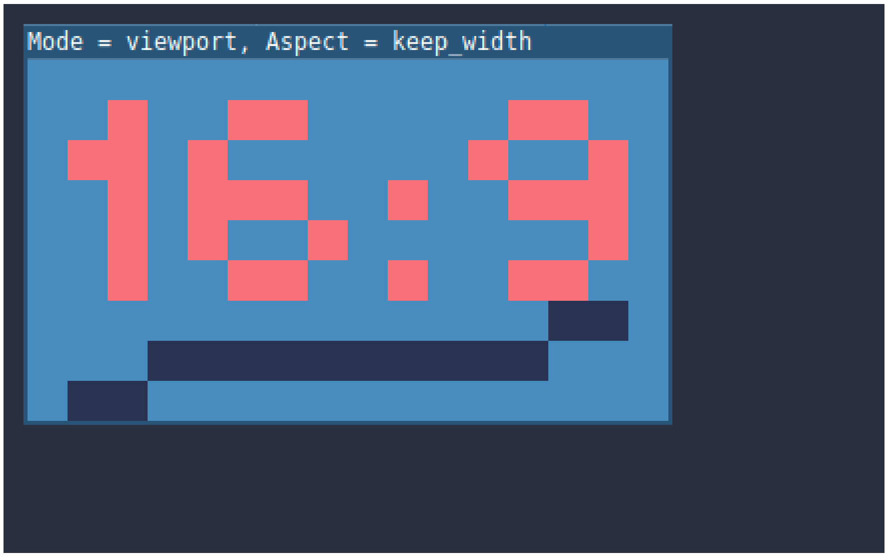
- **Stretch Aspect = Keep**: Keep aspect ratio when stretching the screen. This means that the viewport retains its original size regardless of the screen resolution, and black bars will be added to the top/bottom of the screen ("letterboxing") or the sides ("pillarboxing").

  This is a good option if you know the aspect ratio of your target devices in advance, or if you don't want to handle different aspect ratios.


Mode = viewport, Aspect = keep

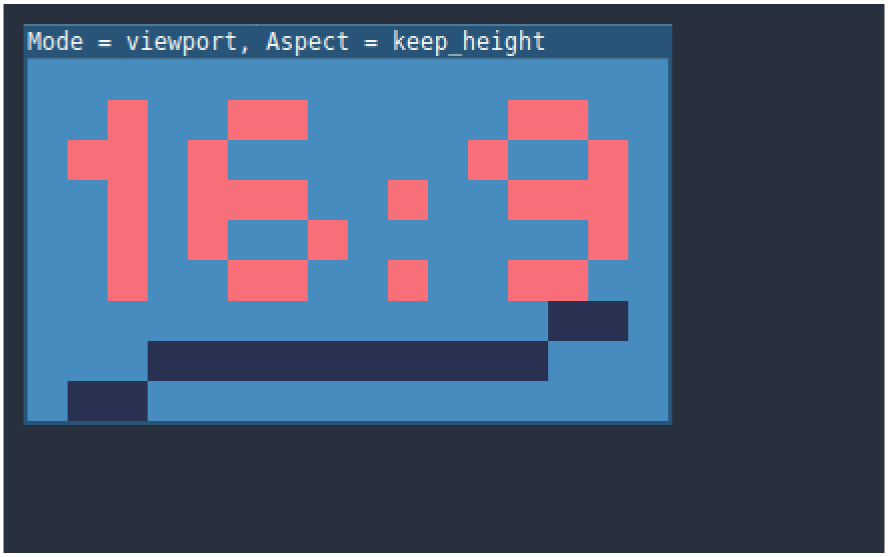
- **Stretch Aspect = Keep Width**: Keep aspect ratio when stretching the screen. If the screen is wider than the base size, black bars are added at the left and right (pillarboxing). But if the screen is taller than the base resolution, the viewport will be grown in the vertical direction (and more content will be visible to the bottom). You can also think of this as "Expand Vertically".

  This is usually the best option for creating GUIs or HUDs that scale, so some controls can be anchored to the bottom (Size and anchors).

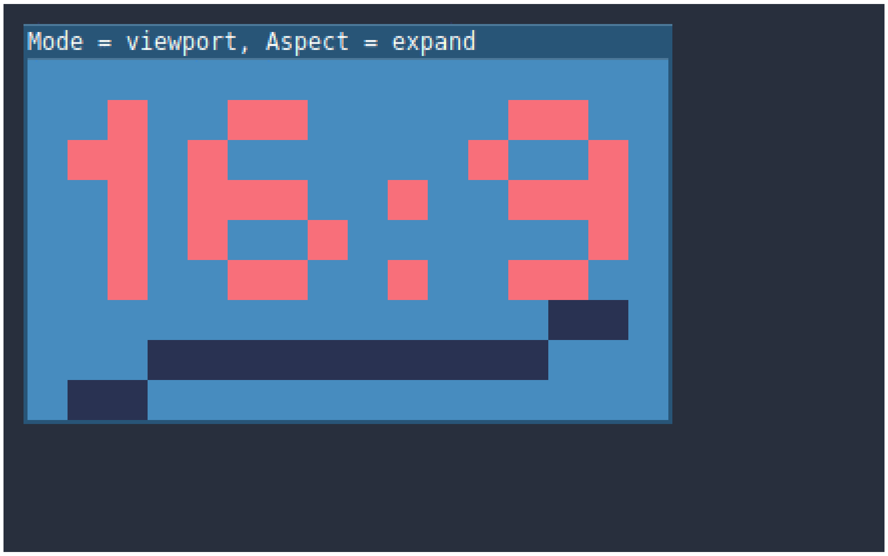
  

- **Stretch Aspect = Keep Height**: Keep aspect ratio when stretching the screen. If the screen is taller than the base size, black bars are added at the top and bottom (letterboxing). But if the screen is wider than the base resolution, the viewport will be grown in the horizontal direction (and more content will be visible to the right). You can also think of this as "Expand Horizontally".

  This is usually the best option for 2D games that scroll horizontally (like runners or platformers).


Mode = viewport, Aspect = keep_height
16:9

- **Stretch Aspect = Expand**: Keep aspect ratio when stretching the screen, but keep neither the base width nor height. Depending on the screen aspect ratio, the viewport will either be larger in the horizontal direction (if the screen is wider than the base size) or in the vertical direction (if the screen is taller than the original size).


Mode = viewport, Aspect = expand
16:9

Tip

To support both portrait and landscape mode with a similar automatically determined scale factor, set your project's base resolution to be a *square* (1:1 aspect ratio) instead of a rectangle. For instance, if you wish to design for 1280×720 as the base resolution but wish to support both portrait and landscape mode, use 720×720 as the project's base window size in the Project Settings.

To allow the user to choose their preferred screen orientation at run-time, remember to set **Display > Window > Handheld > Orientation** to `sensor`.

To configure the stretch aspect at runtime from a script, use the `get_tree().root.content_scale_aspect` property (see Window.content_scale_aspect and the ContentScaleAspect enum).

Stretch Scale

The **Scale** setting allows you to add an extra scaling factor on top of what the **Stretch** options above already provide. The default value of `1.0` means that no additional scaling occurs.

For example, if you set **Scale** to `2.0` and leave **Stretch Mode** on **Disabled**, each unit in your scene will correspond to 2×2 pixels on the screen. This is a good way to provide scaling options for non-game applications.

If **Stretch Mode** is set to **canvas_items**, 2D elements will be scaled relative to the base window size, then multiplied by the **Scale** setting. This can be exposed to players to allow them to adjust the automatically determined scale to their liking, for better accessibility.

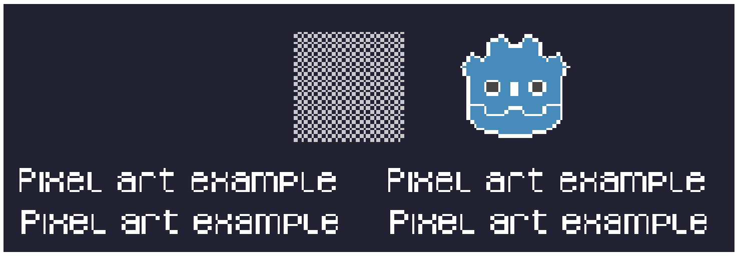
If **Stretch Mode** is set to **viewport**, the viewport's resolution is divided by **Scale**. This makes pixels look larger and reduces rendering resolution (with a given window size), which can improve performance.

To configure the stretch scale at runtime from a script, use the `get_tree().root.content_scale_factor` property (see [Window.content_scale_factor](#)).

Stretch Scale Mode

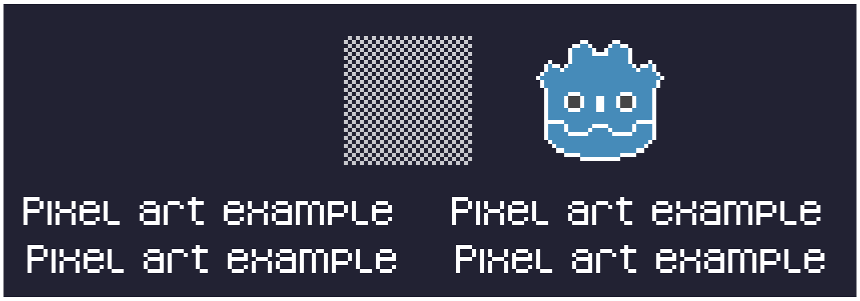
Since Godot 4.2, the **Stretch Scale Mode** setting allows you to constrain the automatically determined scale factor (as well as the manually specified **Stretch Scale** setting) to integer values. By default, this setting is set to `fractional`, which allows any scale factor to be applied (including fractional values such as `2.5`). When set to `integer`, the value is rounded down to the nearest integer. For example, instead of using a scale factor of `2.5`, it would be rounded down to `2.0`. This is useful to prevent distortion when displaying pixel art.

Compare this pixel art which is displayed with the `viewport` stretch mode, with the stretch scale mode set to `fractional`:



Checkerboard doesn't look "even". Line widths in the logo and text varies wildly.

This pixel art is also displayed with the `viewport` stretch mode, but the stretch scale mode is set to `integer` this time:



Checkerboard looks perfectly even. Line widths are consistent.

For example, if your viewport base size is 640×360 and the window size is 1366×768:

- When using `fractional`, the viewport is displayed at a resolution of 1366×768 (scale factor is roughly 2.133×). The entire window space is used. Each pixel in the viewport corresponds to 2.133×2.133 pixels in the displayed area. However, since displays can only display "whole" pixels, this will lead to uneven pixel scaling which results in incorrect appearance of pixel art.
- When using `integer`, the viewport is displayed at a resolution of 1280×720 (scale factor is 2×). The remaining space is filled with black bars on all four sides, so that each pixel in the viewport corresponds to 2×2 pixels in the displayed area.

This setting is effective with any stretch mode. However, when using the `disabled` stretch mode, it will only affect the **Stretch Scale** setting by rounding it *down* to the nearest integer value. This can be used for 3D games that have a pixel art UI, so that the visible area in the 3D viewport doesn't reduce in size (which occurs when using `canvas_items` or `viewport` stretch mode with the `integer` scale mode).

Tip

Games should use the **Exclusive Fullscreen** window mode, as opposed to **Fullscreen** which is designed to prevent Windows from automatically treating the window as if it was exclusive fullscreen.

Fullscreen is meant to be used by GUI applications that want to use per-pixel transparency without a risk of having it disabled by the OS. It achieves this by leaving a 1-pixel line at the bottom of the screen. By contrast, **Exclusive Fullscreen** uses the actual screen size and allows Windows to reduce jitter and input lag for fullscreen games.

When using integer scaling, this is particularly important as the 1-pixel height reduction from the **Fullscreen** mode can cause integer scaling to use a smaller scale factor than expected.

Common use case scenarios

The following settings are recommended to support multiple resolutions and aspect ratios well.

Desktop game

Non-pixel art:

- Set the base window width to `1920` and window height to `1080`. If you have a display smaller than 1920×1080, set **Window Width Override** and **Window Height Override** to lower values to make the window smaller when the project starts.
- Alternatively, if you're targeting high-end devices primarily, set the base window width to `3840` and window height to `2160`. This allows you to provide higher resolution 2D assets, resulting in crisper visuals at the cost of higher memory usage and file sizes. Note that this will make non-mipmapped textures grainy

on low resolution devices, so make sure to follow the instructions described in Reducing aliasing on downsampling.

- Set the stretch mode to `canvas_items`.
- Set the stretch aspect to `expand`. This allows for supporting multiple aspect ratios and makes better use of tall smartphone displays (such as 18:9 or 19:9 aspect ratios).
- Configure Control nodes' anchors to snap to the correct corners using the **Layout** menu.

Pixel art:

- Set the base window size to the viewport size you intend to use. Most pixel art games use viewport sizes between 256×224 and 640×480. 640×360 is a good baseline, as it scales to 1280×720, 1920×1080, 2560×1440, and 3840×2160 without any black bars when using integer scaling. Higher viewport sizes will require using higher resolution artwork, unless you intend to show more of the game world at a given time.
- Set the stretch mode to `viewport`.
- Set the stretch aspect to `keep` to enforce a single aspect ratio (with black bars). As an alternative, you can set the stretch aspect to `expand` to support multiple aspect ratios.
- If using the `expand` stretch aspect, Configure Control nodes' anchors to snap to the correct corners using the **Layout** menu.
- Set the stretch scale mode to `integer`. This prevents uneven pixel scaling from occurring, which makes pixel art not display as intended.

Note

The `viewport` stretch mode provides low-resolution rendering that is then stretched to the final window size. If you are OK with sprites being able to move or rotate in "sub-pixel" positions or wish to have a high resolution 3D viewport, you should use the `canvas_items` stretch mode instead of the `viewport` stretch mode.

Mobile game in landscape mode

Godot is configured to use landscape mode by default. This means you don't need to change the display orientation project setting.

- Set the base window width to `1280` and window height to `720`.
- Alternatively, if you're targeting high-end devices primarily, set the base window width to `1920` and window height to `1080`. This allows you to provide higher resolution 2D assets, resulting in crisper visuals at the cost of higher memory usage and file sizes. Many devices have even higher resolution displays (1440p), but the difference with 1080p is barely visible given the small size of smartphone displays. Note that this will make non-mipmapped textures grainy on low resolution devices, so make sure to follow the instructions described in Reducing aliasing on downsampling.
- Set the stretch mode to `canvas_items`.
- Set the stretch aspect to `expand`. This allows for supporting multiple aspect ratios and makes better use of tall smartphone displays (such as 18:9 or 19:9 aspect ratios).
- Configure Control nodes' anchors to snap to the correct corners using the **Layout** menu.

Tip

To better support tablets and foldable phones (which frequently feature displays with aspect ratios close to 4:3), consider using a base resolution that has a 4:3 aspect ratio while following the rest of the instructions here. For instance, you can set the base window width to `1280` and the base window height to `960`.

Mobile game in portrait mode

- Set the base window width to `720` and window height to `1280`.
- Alternatively, if you're targeting high-end devices primarily, set the base window width to `1080` and window height to `1920`. This allows you to provide higher resolution 2D assets, resulting in crisper visuals at the cost of higher memory usage and file sizes. Many devices have even higher resolution displays (1440p), but the difference with 1080p is barely visible given the small size of smartphone displays. Note that this will make non-mipmapped textures grainy on low resolution devices, so make sure to follow the instructions described in Reducing aliasing on downsampling.
- Set **Display > Window > Handheld > Orientation** to `portrait`.
- Set the stretch mode to `canvas_items`.
- Set the stretch aspect to `expand`. This allows for supporting multiple aspect ratios and makes better use of tall smartphone displays (such as 18:9 or 19:9 aspect ratios).
- Configure Control nodes' anchors to snap to the correct corners using the **Layout** menu.

Tip

To better support tablets and foldable phones (which frequently feature displays with aspect ratios close to 4:3), consider using a base resolution that has a 3:4 aspect ratio while following the rest of the instructions here. For instance, you can set the base window width to `960` and the base window height to `1280`.

Non-game application

- Set the base window width and height to the smallest window size that you intend to target. This is not required, but this ensures that you design your UI with small window sizes in mind.
- Keep the stretch mode to its default value, `disabled`.
- Keep the stretch aspect to its default value, `ignore` (its value won't be used since the stretch mode is `disabled`).
- You can define a minimum window size by calling `get_window().set_min_size()` in a script's `_ready()` function. This prevents the user from resizing the application below a certain size, which could break the UI layout.

Note

Godot doesn't support manually overriding the 2D scale factor yet, so it is not possible to have hiDPI support in non-game applications. Due to this, it is recommended to leave **Allow Hidpi** disabled in non-game applications to allow for the OS to use its low-DPI fallback.

hiDPI support

By default, Godot projects are considered DPI-aware by the operating system. This is controlled by the **Display > Window > Dpi > Allow Hidpi** project setting, which should be left enabled whenever possible. Disabling DPI awareness can break fullscreen behavior on Windows.

Since Godot projects are DPI-aware, they may appear at a very small window size when launching on an hiDPI display (proportionally to the screen resolution). For a game, the most common way to work around this issue is to make them fullscreen by default. Alternatively, you could set the window size in an autoload's `_ready()` function according to the screen size.

To ensure 2D elements don't appear too small on hiDPI displays:

- For games, use the `canvas_items` or `viewport` stretch modes so that 2D elements are automatically resized according to the current window size.
- For non-game applications, use the `disabled` stretch mode and set the stretch scale to a value corresponding to the display scale factor in an autoload's `_ready()` function. The display scale factor is set in the operating system's settings and can be queried using screen_get_scale. This method is currently only implemented on macOS. On other operating systems, you will need to implement a method to guess the display scale factor based on the screen resolution (with a setting to let the user override this if needed). This is the approach currently used by the Godot editor.

The **Allow Hidpi** setting is only effective on Windows and macOS. It's ignored on all other platforms.

Note

The Godot editor itself is always marked as DPI-aware. Running the project from the editor will only be DPI-aware if **Allow Hidpi** is enabled in the Project Settings.

Reducing aliasing on downsampling

If the game has a very high base resolution (e.g. 3840×2160), aliasing might appear when downsampling to something considerably lower like 1280×720.

To resolve this, you can enable mipmaps on all your 2D textures. However, enabling mipmaps will increase memory usage which can be an issue on low-end mobile devices.

Handling aspect ratios

Once scaling for different resolutions is accounted for, make sure that your *user interface* also scales for different aspect ratios. This can be done using anchors and/or containers.

Field of view scaling

The 3D Camera node's **Keep Aspect** property defaults to the **Keep Height** scaling mode (also called *Hor+*). This is usually the best value for desktop games and mobile games in landscape mode, as widescreen displays will automatically use a wider field of view.

However, if your 3D game is intended to be played in portrait mode, it may make more sense to use **Keep Width** instead (also called *Vert-*). This way, smartphones with an aspect ratio taller than 16:9 (e.g. 19:9) will use a *taller* field of view, which is more logical here.

Scaling 2D and 3D elements differently using Viewports

Using multiple Viewport nodes, you can have different scales for various elements. For instance, you can use this to render the 3D world at a low resolution while keeping 2D elements at the native resolution. This can improve performance significantly while keeping the HUD and other 2D elements crisp.

This is done by using the root Viewport node only for 2D elements, then creating a Viewport node to display the 3D world and displaying it using a SubViewportContainer or TextureRect node. There will effectively be two viewports in the final project. One upside of using TextureRect over SubViewportContainer is that it allows enable linear filtering. This makes scaled 3D viewports look better in many cases.

See the 3D viewport scaling demo [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/3d_scaling] for examples.

Fixing jitter, stutter and input lag

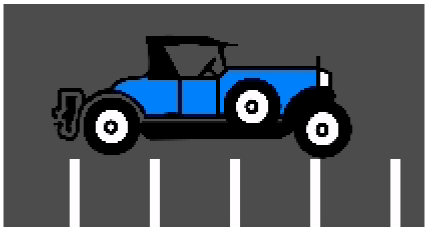
What is jitter, stutter and input lag?

Jitter and *stutter* are two different alterations to visible motion of objects on screen that may affect a game, even when running at full speed. These effects are mostly visible in games where the world moves at a constant speed in a fixed direction, like runners or platformers.

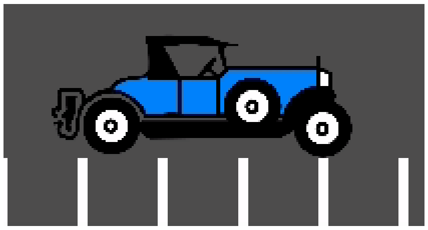
Input lag is unrelated to jitter and stutter, but is sometimes discussed alongside. Input lag refers to visible on-screen delay when performing actions with the mouse, keyboard, controller or touchscreen. It can be related to game code, engine code or external factors (such as hardware). Input lag is most noticeable in games that use the mouse to aim, such as first-person games. Input lag can't be completely eliminated, but it can be reduced in several ways.

Distinguishing between jitter and stutter

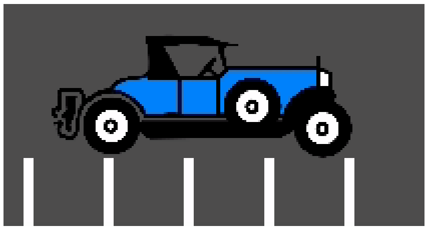
A game running at a normal framerate without exhibiting any effect will appear smooth:



A game exhibiting *jitter* will shake constantly in a very subtle way:



Finally, a game exhibiting *stutter* will appear smooth, but appear to *stop* or *roll back a frame* every few seconds:



Jitter

There can be many causes of jitter, the most typical one happens when the game *physics frequency* (usually 60 Hz) runs at a different resolution than the monitor refresh rate. Check whether your monitor refresh rate is different from 60 Hz.

This is generally not a problem, given that most monitors are 60 Hz, and starting with Godot 3.1, a frame timer was introduced that tries to synchronize with refresh as well as possible.

Sometimes only some objects appear to jitter (character or background). This happens when they are processed in different time sources (one is processed in the physics step while another is processed in the idle step). Godot 3.1 does some improvements to this, from allowing kinematic bodies to be animated in the regular `_process()` loop, to further fixes in the frame timer.

Stutter

Stutter may happen due to two different reasons. The first, and most obvious one, is the game not being able to keep full framerate performance. Solving this is game specific and will require optimization.

The second is more complicated, because it is often not associated to the engine or game but the underlying operating system. Here is some information regarding stutter on different OSs.

On platforms that support disabling V-Sync, stuttering can be made less noticeable by disabling V-Sync in the project settings. This will however cause tearing to appear, especially on monitors with low refresh rates. If your monitor supports it, consider enabling variable refresh rate (G-Sync/FreeSync) while leaving V-Sync enabled. This avoids mitigating some forms of stuttering without introducing tearing.

Forcing your graphics card to use the maximum performance profile can also help reduce stuttering, at the cost of increased GPU power draw.

Windows

Windows is known to cause stutter in windowed games. This mostly depends on the hardware installed, drivers version and processes running in parallel (e.g. having many browser tabs open may cause stutter in a running game). To avoid this, starting with 3.1, Godot raises the game priority to "Above Normal". This helps considerably but may not completely eliminate stutter.

Eliminating this completely requires giving your game full privileges to become "time critical", which is not advised. Some games may do it, but it is advised to learn to live with this problem, as it is common for Windows games and most users won't play games windowed (games that are played in a window, e.g. puzzle games, will usually not exhibit this problem anyway).

For fullscreen, Windows gives special priority to the game so stutter is no longer visible and very rare. This is how most games are played.

When using a mouse with a polling rate of 1,000 Hz or more, consider using a fully up-to-date Windows 11 installation which comes with fixes related to high CPU utilization with high polling rate mice. These fixes are not available in Windows 10 and older versions.

Tip

Games should use the **Exclusive Fullscreen** window mode, as opposed to **Fullscreen** which is designed to prevent Windows from automatically treating the window as if it was exclusive fullscreen.

Fullscreen is meant to be used by GUI applications that want to use per-pixel transparency without a risk of having it disabled by the OS. It achieves this by leaving a 1-pixel line at the bottom of the screen. By contrast, **Exclusive Fullscreen** uses the actual screen size and allows Windows to reduce jitter and input lag for fullscreen games.

Linux

Stutter may be visible on desktop Linux, but this is usually associated with different video drivers and compositors. Some compositors may also trigger this problem (e.g. KWin), so it is advised to try using a different one to rule it out as the cause. Some window managers such as KWin and Xfwm allow you to manually disable compositing, which can improve performance (at the cost of tearing).

There is no workaround for driver or compositor stuttering, other than reporting it as an issue to the driver or compositor developers. Stutter may be more present when playing in windowed mode as opposed to fullscreen, even with compositing disabled.

Feral GameMode [https://github.com/FeralInteractive/gamemode] can be used to automatically apply optimizations (such as forcing the GPU performance profile) when running specific processes.

macOS

Generally, macOS is stutter-free, although recently some bugs were reported when running on fullscreen (this is a macOS bug). If you have a machine exhibiting this behavior, please let us know.

Android

Generally, Android is stutter and jitter-free because the running activity gets all the priority. That said, there may be problematic devices (older Kindle Fire is known to be one). If you see this problem on Android, please let us know.

iOS

iOS devices are generally stutter-free, but older devices running newer versions of the operating system may exhibit problems. This is generally unavoidable.

Input lag

Project configuration

On platforms that support disabling V-Sync, input lag can be made less noticeable by disabling V-Sync in the project settings. This will however cause tearing to appear, especially on monitors with low refresh rates. It's suggested to make V-Sync available as an option for players to toggle.

When using the Forward+ or Mobile rendering methods, another way to reduce visual latency when V-Sync is enabled is to use double-buffered V-Sync instead of the default triple-buffered V-Sync. Since Godot 4.3, this can be achieved by reducing the **Display > Window > V-Sync > Swapchain Image Count** project setting to `2`. The downside of using double buffering is that framerate will be less stable if the display refresh rate can't be reached due to a CPU or GPU bottleneck. For instance, on a 60 Hz display, if the framerate would normally drop to 55 FPS during gameplay with triple buffering, it will have to drop down to 30 FPS momentarily with double buffering (and then go back to 60 FPS when possible). As a result, double-buffered V-Sync is only recommended if you can *consistently* reach the display refresh rate on the target hardware.

Increasing the number of physics iterations per second can also reduce physics-induced input latency. This is especially noticeable when using physics interpolation (which improves smoothness but increases latency). To do so, set **Physics > Common > Physics Ticks Per Second** to a value higher than the default `60`, or set `Engine.physics_ticks_per_second` at run-time in a script. Values that are a multiple of the monitor refresh rate (typically `60`) work best when physics interpolation is disabled, as they will avoid jitter. This means values such as `120`, `180` and `240` are good starting points. As a bonus, higher physics FPSes make tunneling and physics instability issues less likely to occur.

The downside of increasing physics FPS is that CPU usage will increase, which can lead to performance bottlenecks in games that have heavy physics simulation code. This can be alleviated by increasing physics FPS only in situations where low latency is critical, or by letting players adjust physics FPS to match their hardware. However, different physics FPS will lead to different outcomes in physics simulation, even when `delta` is consistently used in your game logic. This can give certain players an advantage over others. Therefore, allowing the player to change the physics FPS themselves should be avoided for competitive multiplayer games.

Lastly, you can disable input buffering on a per-rendered frame basis by calling `Input.set_use_accumulated_input(false)` in a script. This will make it so the `_input()` and `_unhandled_input()` functions in your scripts are called on every input, rather than accumulating inputs and waiting for a frame to be rendered. Disabling input accumulation will increase CPU usage, so it should be done with caution.

Tip

On any Godot project, you can use the `--disable-vsync` command line argument to forcibly disable V-Sync. Since Godot 4.2, `--max-fps <fps>` can also be used to set a FPS limit (`0` is unlimited). These arguments can be used at the same time.

Hardware/OS-specific

If your monitor supports it, consider enabling variable refresh rate (G-Sync/FreeSync) while leaving V-Sync enabled, then cap the framerate in the project settings to a slightly lower value than your monitor's maximum refresh rate as per this page [https://blurbusters.com/howto-low-lag-vsync-on/]. For example, on a 144 Hz monitor, you can set the project's framerate cap to `141`. This may be counterintuitive at first, but capping the FPS below the maximum refresh rate range ensures that the OS never has to wait for vertical blanking to finish. This leads to *similar* input lag as V-Sync disabled with the same framerate cap (usually less than 1 ms greater), but without any tearing.

This can be done by changing the **Application > Run > Max FPS** project setting or assigning `Engine.max_fps` at run-time in a script.

On some platforms, you can also opt into a low-latency mode in the graphics driver options (such as the NVIDIA Control Panel on Windows). The **Ultra** setting will give you the lowest possible latency, at the cost of slightly lower average framerates. Forcing the GPU to use the maximum performance profile can also further reduce input lag, at the cost of higher power consumption (and resulting heat/fan noise).

Finally, make sure your monitor is running at its highest possible refresh rate in the OS' display settings.

Also, ensure that your mouse is configured to use its highest polling rate (typically 1,000 Hz for gaming mice, sometimes more). High USB polling rates can however result in high CPU usage, so 500 Hz may be a safer bet on low-end CPUs. If your mouse offers multiple DPI settings, consider also using the highest possible setting and reducing in-game sensitivity to reduce mouse latency [https://www.youtube.com/watch?v=6AoRfv9W110].

On Linux, disabling compositing in window managers that allow it (such as KWin or Xfwm) can reduce input lag significantly.

Reporting jitter, stutter or input lag problems

If you are reporting a stutter or jitter problem (opening an issue) not caused by any of the above reasons, please specify very clearly all the information possible about device, operating system, driver versions, etc. This may help to better troubleshoot it.

If you are reporting input lag problems, please include a capture made with a high speed camera (such as your phone's slow motion video mode). The capture **must** have both the screen and the input device visible so that the number of frames between an input and the on-screen result can be counted. Also, make sure to mention your monitor's refresh rate and your input device's polling rate (especially for mice).

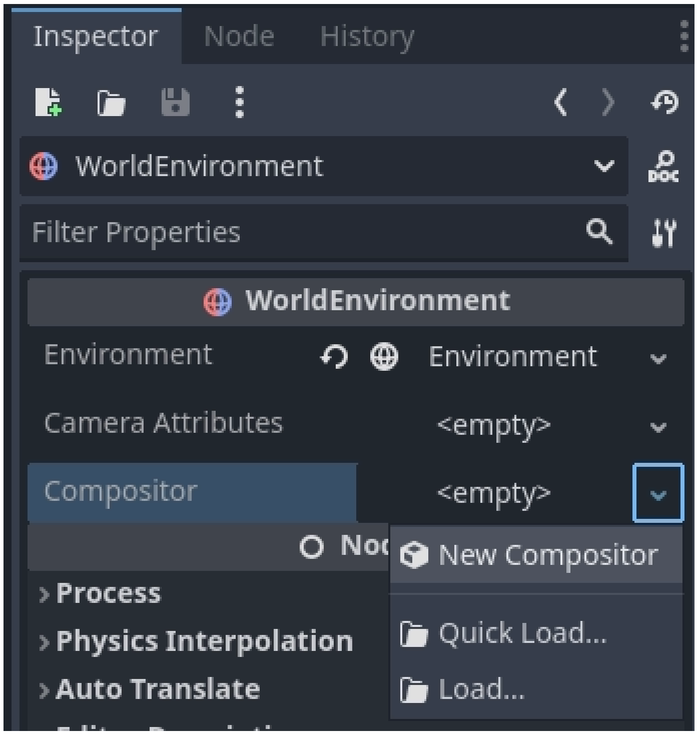
Also, make sure to use the correct term (jitter, stutter, input lag) based on the exhibited behavior. This will help understand your issue much faster. Provide a project that can be used to reproduce the issue, and if possible, include a screen capture demonstrating the bug.

The Compositor

The compositor is a new feature in Godot 4 that allows control over the rendering pipeline when rendering the contents of a Viewport.

It can be configured on a WorldEnvironment node where it applies to all Viewports, or it can be configured on a Camera3D and apply only to the Viewport using that camera.

The Compositor resource is used to configure the compositor. To get started, simply create a new compositor on the appropriate node:


Inspector
Node
History
WorldEnvironment
Filter Properties
WorldEnvironment
Environment
Environment
Camera Attributes
<empty>
Compositor
<empty>
Noc
New Compositor
Quick Load...
Load...
Process
Physics Interpolation
Auto Translate

Note

The compositor is currently a feature that is only supported by the Mobile and Forward+ renderers.

Compositor effects

Compositor effects allow you to insert additional logic into the rendering pipeline at various stages. This is an advanced feature that requires a high level of understanding of the rendering pipeline to use to its best advantage.

As the core logic of the compositor effect is called from the rendering pipeline it is important to note that this logic will thus run within the thread on which rendering takes place. Care needs to be taken to ensure we don't run into threading issues.

To illustrate how to use compositor effects we'll create a simple post processing effect that allows you to write your own shader code and apply this full screen through a compute shader. You can find the finished demo project here [https://github.com/godotengine/godot-demo-projects/tree/master/compute/post_shader].

We start by creating a new script called `post_process_shader.gd`. We'll make this a tool script so we can see the compositor effect work in the editor. We need to extend our node from CompositorEffect. We must also give our script a class name.

```
@tool
extends CompositorEffect
class_name PostProcessShader
```

Next we're going to define a constant for our shader template code. This is the boilerplate code that makes our compute shader work.

```
const template_shader : String = "#version 450

// Invocations in the (x, y, z) dimension
layout(local_size_x = 8, local_size_y = 8, local_size_z = 1)
in;

layout(rgba16f, set = 0, binding = 0) uniform image2D
```

```
color_image;

// Our push constant
layout(push_constant, std430) uniform Params {
    vec2 raster_size;
    vec2 reserved;
} params;

// The code we want to execute in each invocation
void main() {
    ivec2 uv = ivec2(gl_GlobalInvocationID.xy);
    ivec2 size = ivec2(params.raster_size);

    if (uv.x >= size.x || uv.y >= size.y) {
        return;
    }

    vec4 color = imageLoad(color_image, uv);

    #COMPUTE_CODE

    imageStore(color_image, uv, color);
}"
```

For more information on how compute shaders work, please check Using compute shaders.

The important bit here is that for every pixel on our screen, our `main` function is executed and inside of this we load the current color value of our pixel, execute our user code, and write our modified color back to our color image.

`#COMPUTE_CODE` gets replaced by our user code.

In order to set our user code, we need an export variable. We'll also define a few script variables we'll be using:

```
@export_multiline var shader_code : String = "":
    set(value):
        mutex.lock()
        shader_code = value
        shader_is_dirty = true
        mutex.unlock()

var rd : RenderingDevice
var shader : RID
var pipeline : RID

var mutex : Mutex = Mutex.new()
var shader_is_dirty : bool = true
```

Note the use of a Mutex in our code. Most of our implementation gets called from the rendering engine and thus runs within our rendering thread.

We need to ensure that we set our new shader code, and mark our shader code as dirty, without our render thread accessing this data at the same time.

Next we initialize our effect.

```
# Called when this resource is constructed.
func _init():
    effect_callback_type =
EFFECT_CALLBACK_TYPE_POST_TRANSPARENT
    rd = RenderingServer.get_rendering_device()
```

The main thing here is setting our `effect_callback_type` which tells the rendering engine at what stage of the render pipeline to call our code.

Note

Currently we only have access to the stages of the 3D rendering pipeline!

We also get a reference to our rendering device, which will come in very handy.

We also need to clean up after ourselves, for this we react to the `NOTIFICATION_PREDELETE` notification:

```
# System notifications, we want to react on the notification
that
# alerts us we are about to be destroyed.
func _notification(what):
    if what == NOTIFICATION_PREDELETE:
        if shader.is_valid():
            # Freeing our shader will also free any dependents
such as the pipeline!
            rd.free_rid(shader)
```

Note that we do not use our mutex here even though we create our shader inside of our render thread. The methods on our rendering server are thread safe and `free_rid` will be postponed cleaning up the shader until after any frames currently being rendered are finished.

Also note that we are not freeing our pipeline. The rendering device does dependency tracking and as the pipeline is dependent on the shader, it will be automatically freed when the shader is destructed.

From this point onwards our code will run on the rendering thread.

Our next step is a helper function that will recompile the shader if the user code was changed.

```
# Check if our shader has changed and needs to be recompiled.
func _check_shader() -> bool:
    if not rd:
        return false

    var new_shader_code : String = ""

    # Check if our shader is dirty.
    mutex.lock()
    if shader_is_dirty:
        new_shader_code = shader_code
        shader_is_dirty = false
    mutex.unlock()

    # We don't have a (new) shader?
    if new_shader_code.is_empty():
        return pipeline.is_valid()

    # Apply template.
    new_shader_code = template_shader.replace("#COMPUTE_CODE", new_shader_code);

    # Out with the old.
    if shader.is_valid():
        rd.free_rid(shader)
        shader = RID()
        pipeline = RID()

    # In with the new.
    var shader_source : RDShaderSource = RDShaderSource.new()
    shader_source.language = RenderingDevice.SHADER_LANGUAGE_GLSL
    shader_source.source_compute = new_shader_code
    var shader_spirv : RDShaderSPIRV = rd.shader_compile_spirv_from_source(shader_source)

    if shader_spirv.compile_error_compute != "":
        push_error(shader_spirv.compile_error_compute)
        push_error("In: " + new_shader_code)
        return false

    shader = rd.shader_create_from_spirv(shader_spirv)
    if not shader.is_valid():
        return false

    pipeline = rd.compute_pipeline_create(shader)
    return pipeline.is_valid()
```

At the top of this method we again use our mutex to protect accessing our user shader code and our is dirty flag. We make a local copy of the user shader code if our user shader code is dirty.

If we don't have a new code fragment, we return true if we already have a valid pipeline.

If we do have a new code fragment we embed it in our template code and then compile it.

Warning

The code shown here compiles our new code in runtime. This is great for prototyping as we can immediately see the effect of the changed shader.

This prevents precompiling and caching this shader which may be an issues on some platforms such as consoles. Note that the demo project comes with an alternative example where a `glsl` file contains the entire compute shader and this is used. Godot is able to precompile and cache the shader with this approach.

Finally we need to implement our effect callback, the rendering engine will call this at the right stage of rendering.

```
# Called by the rendering thread every frame.
func _render_callback(p_effect_callback_type, p_render_data):
    if rd and p_effect_callback_type ==
EFFECT_CALLBACK_TYPE_POST_TRANSPARENT and _check_shader():
        # Get our render scene buffers object, this gives us
access to our render buffers.
        # Note that implementation differs per renderer hence
the need for the cast.
        var render_scene_buffers : RenderSceneBuffersRD =
p_render_data.get_render_scene_buffers()
        if render_scene_buffers:
            # Get our render size, this is the 3D render
resolution!
            var size = render_scene_buffers.get_internal_size()
            if size.x == 0 and size.y == 0:
                return

            # We can use a compute shader here
            var x_groups = (size.x - 1) / 8 + 1
            var y_groups = (size.y - 1) / 8 + 1
            var z_groups = 1

            # Push constant
            var push_constant : PackedFloat32Array =
PackedFloat32Array()
            push_constant.push_back(size.x)
            push_constant.push_back(size.y)
            push_constant.push_back(0.0)
            push_constant.push_back(0.0)

            # Loop through views just in case we're doing
stereo rendering. No extra cost if this is mono.
            var view_count =
render_scene_buffers.get_view_count()
            for view in range(view_count):
                # Get the RID for our color image, we will be
reading from and writing to it.
                var input_image =
render_scene_buffers.get_color_layer(view)

                # Create a uniform set, this will be cached,
the cache will be cleared if our viewports configuration is
changed.
                var uniform : RDUniform = RDUniform.new()
                uniform.uniform_type =
RenderingDevice.UNIFORM_TYPE_IMAGE
                uniform.binding = 0
                uniform.add_id(input_image)
                var uniform_set =
UniformSetCacheRD.get_cache(shader, 0, [ uniform ])

                # Run our compute shader.
                var compute_list := rd.compute_list_begin()

rd.compute_list_bind_compute_pipeline(compute_list, pipeline)
                rd.compute_list_bind_uniform_set(compute_list,
uniform_set, 0)
                rd.compute_list_set_push_constant(compute_list,
push_constant.to_byte_array(), push_constant.size() * 4)
                rd.compute_list_dispatch(compute_list,
x_groups, y_groups, z_groups)
                rd.compute_list_end()
```

At the start of this method we check if we have a rendering device, if our callback type is the correct one, and check if we have our shader.

Note

The check for the effect type is only a safety mechanism. We've set this in our `_init` function, however it is possible for the user to change this in the UI.

Our `p_render_data` parameter gives us access to an object that holds data specific to the frame we're currently rendering. We're currently only interested in our render scene buffers, which provide us access to all the internal buffers used by the rendering engine. Note that we cast this to RenderSceneBuffersRD to expose the full API to this data.

Next we obtain our `internal size` which is the resolution of our 3D render buffers before they are upscaled (if applicable), upscaling happens after our post processes have run.

From our internal size we calculate our group size, see our local size in our template shader.

We also populate our push constant so our shader knows our size. Godot does not support structs here **yet** so we use a `PackedFloat32Array` to store this data into. Note that we have to pad this array with a 16 byte alignment. In other words, the length of our array needs to be a multiple of 4.

Now we loop through our views, this is in case we're using multiview rendering which is applicable for stereo rendering (XR). In most cases we will only have one view.

Note

There is no performance benefit to use multiview for post processing here, handling the views separately like this will still enable the GPU to use parallelism if beneficial.

Next we obtain the color buffer for this view. This is the buffer into which our 3D scene has been rendered.

We then prepare a uniform set so we can communicate the color buffer to our shader.

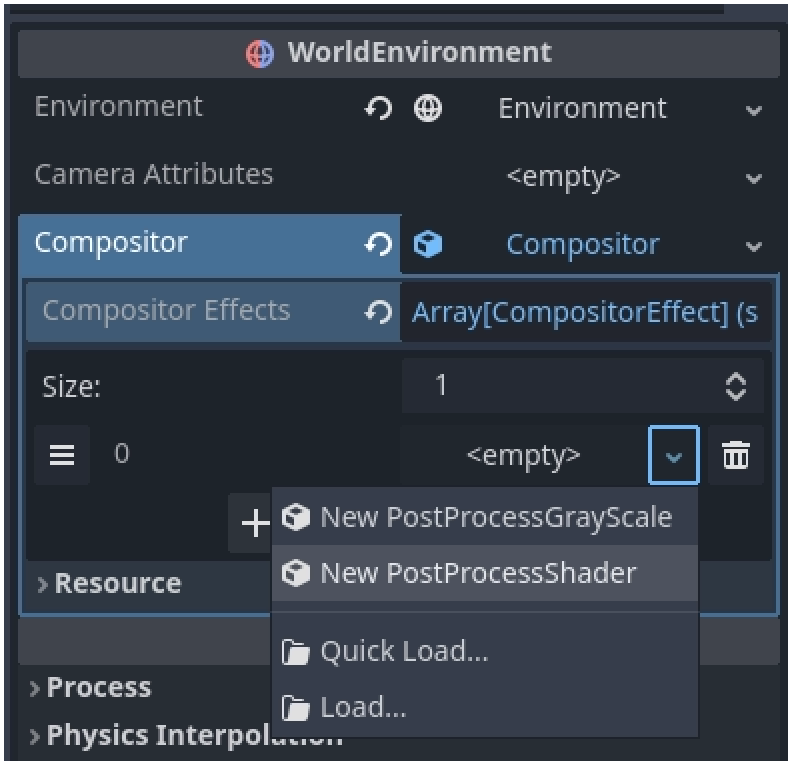
Note the use of our UniformSetCacheRD cache which ensures we can check for our uniform set each frame. As our color buffer can change from frame to frame and our uniform cache will automatically clean up uniform sets when buffers are freed, this is the safe way to ensure we do not leak memory or use an outdated set.

Finally we build our compute list by binding our pipeline, binding our uniform set, pushing our push constant data, and calling dispatch for our groups.

With our compositor effect completed, we now need to add it to our compositor.

On our compositor we expand the compositor effects property and press `Add Element`.

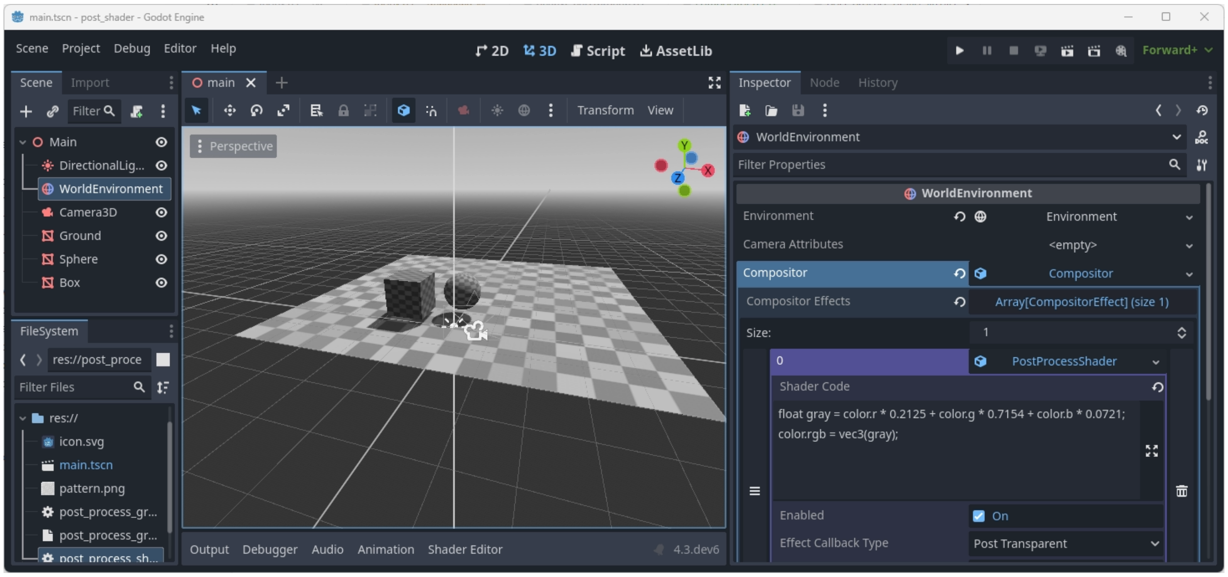
Now we can add our compositor effect:


WorldEnvironment
Environment
Environment
Camera Attributes
<empty>
Compositor
Compositor
Compositor Effects
Array[CompositorEffect] (s
Size:
1
0
<empty>
New PostProcessGrayScale
New PostProcessShader
Quick Load...
Load...
Resource
Process
Physics Interpol

After selecting our `PostProcessShader` we need to set our user shader code:

```
float gray = color.r * 0.2125 + color.g * 0.7154 + color.b *
0.0721;
color.rgb = vec3(gray);
```

With that all done, our output is in grayscale.


main.tscn - post_shader - Godot Engine
Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Forward+
Scene
Import
main
Transform
View
Perspective
Main
DirectionalLig...
WorldEnvironment
Camera3D
Ground
Sphere
Box
FileSystem
res://post_proce
Filter Files
res://
icon.svg
main.tscn
pattern.png
post_process_gr...
post_process_gr...
post_process_sh...
Output
Debugger
Audio
Animation
Shader Editor
4.3.dev6
Inspector
Node
History
Filter Properties

WorldEnvironment

Environment: Environment

Camera Attributes: <empty>

Compositor: Compositor

Compositor Effects: Array[CompositorEffect] (size 1)

Size: 1

0: PostProcessShader

Shader Code:

```
float gray = color.r * 0.2125 + color.g * 0.7154 + color.b * 0.0721;
color.rgb = vec3(gray);
```

Enabled: ☑ On

Effect Callback Type: Post Transparent

Note

For a more advanced example of post effects, check out the Radial blur based sky rays [https://github.com/BastiaanOlij/RERadialSunRays] example project created by Bastiaan Olij.

Scripting

This section covers programming languages and core features to code your games in Godot.

Here, you will find information that is not already covered in more specific sections. For instance, to learn about inputs, we recommend you to read Inputs.

Programming languages

The sections below each focus on a given programming language.

- GDScript
- C#/.NET
- GDExtension

Core features

Some features are specific to the engine and are available in all supported languages. Whether you code in GDScript, C#, or another language, the pages below will help you make the most of Godot.

- How to read the Godot API
- Debug
- Idle and Physics Processing
- Groups
- Nodes and scene instances
- Overridable functions
- Cross-language scripting
- Creating script templates
- Evaluating expressions
- Change scenes manually

- Instancing with signals
- Pausing games and process mode
- File system
- Resources
- Singletons (Autoload)
- Using SceneTree
- Scene Unique Nodes

GDScript

- GDScript reference
- GDScript: An introduction to dynamic languages
- GDScript exported properties
- GDScript documentation comments
- GDScript style guide
- Static typing in GDScript
- GDScript warning system
- GDScript format strings

See also

See GDScript grammar if you are interested in writing a third-party tool that interacts with GDScript, such as a linter or formatter.

GDScript reference

GDScript is a high-level, object-oriented [https://en.wikipedia.org/wiki/Object-oriented_programming], imperative [https://en.wikipedia.org/wiki/Imperative_programming], and gradually typed [https://en.wikipedia.org/wiki/Gradual_typing] programming language built for Godot. It uses an indentation-based syntax similar to languages like Python [https://en.wikipedia.org/wiki/Python_%28programming_language%29]. Its goal is to be optimized for and tightly integrated with Godot Engine, allowing great flexibility for content creation and integration.

GDScript is entirely independent from Python and is not based on it.

History

Note

Documentation about GDScript's history has been moved to the Frequently Asked Questions.

Example of GDScript

Some people can learn better by taking a look at the syntax, so here's an example of how GDScript looks.

```
# Everything after "#" is a comment.
# A file is a class!

# (optional) icon to show in the editor dialogs:
@icon("res://path/to/optional/icon.svg")

```

```
# (optional) class definition:
class_name MyClass

# Inheritance:
extends BaseClass


# Member variables.
var a = 5
var s = "Hello"
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var other_dict = {key = "value", other_key = 2}
var typed_var: int
var inferred_type := "String"

# Constants.
const ANSWER = 42
const THE_NAME = "Charly"

# Enums.
enum {UNIT_NEUTRAL, UNIT_ENEMY, UNIT_ALLY}
enum Named {THING_1, THING_2, ANOTHER_THING = -1}

# Built-in vector types.
var v2 = Vector2(1, 2)
var v3 = Vector3(1, 2, 3)


# Functions.
func some_function(param1, param2, param3):
    const local_const = 5

    if param1 < local_const:
        print(param1)
    elif param2 > 5:
        print(param2)
    else:
        print("Fail!")

    for i in range(20):
        print(i)
    while param2 != 0:
        param2 -= 1

    match param3:
        3:
            print("param3 is 3!")
        _:
            print("param3 is not 3!")

    var local_var = param1 + 3
    return local_var


# Functions override functions with the same name on the
base/super class.
# If you still want to call them, use "super":
func something(p1, p2):
    super(p1, p2)


# It's also possible to call another function in the super
class:
func other_something(p1, p2):
    super.something(p1, p2)


# Inner class
class Something:
    var a = 10


# Constructor
func _init():
    print("Constructed!")
    var lv = Something.new()
    print(lv.a)
```

If you have previous experience with statically typed languages such as C, C++, or C# but never used a dynamically typed one before, it is advised you read this tutorial: GDScript: An introduction to dynamic languages.

Language

In the following, an overview is given to GDScript. Details, such as which methods are available to arrays or other objects, should be looked up in the linked class descriptions.

Identifiers

Any string that restricts itself to alphabetic characters (`a` to `z` and `A` to `Z`), digits (`0` to `9`) and `_` qualifies as an identifier. Additionally, identifiers must not begin with a digit. Identifiers are case-sensitive (`foo` is different from `FOO`).

Identifiers may also contain most Unicode characters part of UAX#31 [https://www.unicode.org/reports/tr31/]. This allows you to use identifier names written in languages other than English. Unicode characters that are considered "confusable" for ASCII characters and emoji are not allowed in identifiers.

Keywords

The following is the list of keywords supported by the language. Since keywords are reserved words (tokens), they can't be used as identifiers. Operators (like `in`, `not`, `and` or `or`) and names of built-in types as listed in the following sections are also reserved.

Keywords are defined in the GDScript tokenizer [https://github.com/godotengine/godot/blob/master/modules/gdscript/gdscript_tokenizer.cpp] in case you want to take a look under the hood.

Keyword	**Description**
if	See if/else/elif.
elif	See if/else/elif.
else	See if/else/elif.
for	See for.
while	See while.
match	See match.
when	Used by pattern guards in `match` statements.
break	Exits the execution of the current `for` or `while` loop.
continue	Immediately skips to the next iteration of the `for` or `while` loop.
pass	Used where a statement is required syntactically but execution of code is undesired, e.g. in empty functions.
return	Returns a value from a function.

Keyword	Description
class	Defines an inner class. See Inner classes.
class_name	Defines the script as a globally accessible class with the specified name. See Registering named classes.
extends	Defines what class to extend with the current class.
is	Tests whether a variable extends a given class, or is of a given built-in type.
in	Tests whether a value is within a string, array, range, dictionary, or node. When used with `for`, it iterates through them instead of testing.
as	Cast the value to a given type if possible.
self	Refers to current class instance.
super	Resolves the scope of the parent method. See Inheritance.
signal	Defines a signal.
func	Defines a function.

Keyword	Description
static	Defines a static function or a static member variable.
const	Defines a constant.
enum	Defines an enum.
var	Defines a variable.
breakpoint	Editor helper for debugger breakpoints. Unlike breakpoints created by clicking in the gutter, `breakpoint` is stored in the script itself. This makes it persistent across different machines when using version control.
preload	Preloads a class or variable. See Classes as resources.
await	Waits for a signal or a coroutine to finish. See Awaiting signals or coroutines.
yield	Previously used for coroutines. Kept as keyword for transition.
assert	Asserts a condition, logs error on failure. Ignored in non-debug builds. See Assert keyword.

Keyword	Description
void	Used to represent that a function does not return any value.
PI	PI constant.
TAU	TAU constant.
INF	Infinity constant. Used for comparisons and as result of calculations.
NAN	NAN (not a number) constant. Used as impossible result from calculations.

Operators

The following is the list of supported operators and their precedence. All binary operators are left-associative [https://en.wikipedia.org/wiki/Operator_associativity], including the `**` operator. This means that `2 ** 2 ** 3` is equal to `(2 ** 2) ** 3`. Use parentheses to explicitly specify precedence you need, for example `2 ** (2 ** 3)`. The ternary `if/else` operator is right-associative.

Operator	Description
`( )`	Grouping (highest priority)

Operator	**Description**
	Parentheses are not really an operator, but allow you to explicitly specify the precedence of an operation.
`x[index]`	Subscription
`x.attribute`	Attribute reference
`foo()`	Function call
`await x`	Awaiting signals or coroutines
`x is Node` `x is not Node`	Type checking See also is_instance_of() function.
`x ** y`	Power Multiplies `x` by itself `y` times, similar to calling pow() function.
`~x`	Bitwise NOT
`+x` `-x`	Identity / Negation

Operator	Description

`x * y`
`x / y`
`x % y`
	Multiplication / Division / Remainder
	The `%` operator is additionally used for format strings.
	Note: These operators have the same behavior as C++, which may be unexpected for users coming from Python, JavaScript, etc. See a detailed note after the table.

`x + y` `x - y`	Addition (or Concatenation) / Subtraction
`x << y` `x >> y`	Bit shifting
`x & y`	Bitwise AND
`x ^ y`	Bitwise XOR
`x | y`	Bitwise OR

`x == y`
`x != y`
`x < y`
`x > y`
	Comparison
	See a detailed note after the table.

Operator	Description
`x <= y` `x >= y`	
`x in y` `x not in y`	Inclusion checking `in` is also used with the for keyword as part of the syntax.
`not x` `!x`	Boolean NOT and its unrecommended alias
`x and y` `x && y`	Boolean AND and its unrecommended alias
`x or y` `x || y`	Boolean OR and its unrecommended alias
`true_expr if cond else false_expr`	Ternary if/else
`x as Node`	Type casting

`x = y`
`x += y`
`x -= y`
`x *= y`
`x /= y`
	Assignment (lowest priority)
	You cannot use an assignment operator inside an expression.

Operator	Description

`x **= y`
`x %= y`
`x &= y`
`x |= y`
`x ^= y`
`x <<= y`
`x >>= y`	

Note

The behavior of some operators may differ from what you expect:

1. If both operands of the `/` operator are int, then integer division is performed instead of fractional. For example `5 / 2 == 2`, not `2.5`. If this is not desired, use at least one float literal (`x / 2.0`), cast (`float(x) / y`), or multiply by `1.0` (`x * 1.0 / y`).
2. The `%` operator is only available for ints, for floats use the fmod() function.
3. For negative values, the `%` operator and `fmod()` use truncation [https://en.wikipedia.org/wiki/Truncation] instead of rounding towards negative infinity. This means that the remainder has a sign. If you need the remainder in a mathematical sense, use the posmod() and fposmod() functions instead.
4. The `==` and `!=` operators sometimes allow you to compare values of different types (for example, `1 == 1.0` is true), but in other cases it can cause a runtime error. If you're not sure about the types of the operands, you can safely use the is_same() function (but note that it is more strict about types and references). To compare floats, use the is_equal_approx() and is_zero_approx() functions instead.

Literals

Example(s)	Description
`null`	Null value
`false`, `true`	Boolean values
`45`	Base 10 integer
`0x8f51`	Base 16 (hexadecimal) integer
`0b101010`	Base 2 (binary) integer
`3.14`, `58.1e-10`	Floating-point number (real)
`"Hello"`, `'Hi'`	Regular strings
`"""Hello"""`, `'''Hi'''`	Triple-quoted regular strings
`r"Hello"`, `r'Hi'`	Raw strings
`r"""Hello"""`, `r'''Hi'''`	Triple-quoted raw strings
`&"name"`	StringName

`^"Node/Label"`	NodePath

There are also two constructs that look like literals, but actually are not:

Example	Description
`$NodePath`	Shorthand for `get_node("NodePath")`
`%UniqueNode`	Shorthand for `get_node("%UniqueNode")`

Integers and floats can have their numbers separated with _ to make them more readable. The following ways to write numbers are all valid:

```
12_345_678  # Equal to 12345678.
3.141_592_7  # Equal to 3.1415927.
0x8080_0000_ffff  # Equal to 0x80800000ffff.
0b11_00_11_00  # Equal to 0b11001100.
```

Regular string literals can contain the following escape sequences:

Escape sequence	Expands to
`\n`	Newline (line feed)
`\t`	Horizontal tab character

`\r`	Carriage return
`\a`	Alert (beep/bell)
`\b`	Backspace
`\f`	Formfeed page break
`\v`	Vertical tab character
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash
`\uXXXX`	UTF-16 Unicode codepoint `XXXX` (hexadecimal, case-insensitive)
`\UXXXXXX`	UTF-32 Unicode codepoint `XXXXXX` (hexadecimal, case-insensitive)

There are two ways to represent an escaped Unicode character above `0xFFFF`:

- as a UTF-16 surrogate pair [https://en.wikipedia.org/wiki/UTF-16#Code_points_from_U+010000_to_U+10FFFF] `\uXXXX\uXXXX`.
- as a single UTF-32 codepoint `\UXXXXXX`.

Also, using `\` followed by a newline inside a string will allow you to continue it in the next line, without inserting a newline character in the string itself.

A string enclosed in quotes of one type (for example `"`) can contain quotes of another type (for example `'`) without escaping. Triple-quoted strings allow you to avoid escaping up to two consecutive quotes of the same type (unless they are adjacent to the string edges).

Raw string literals always encode the string as it appears in the source code. This is especially useful for regular expressions. A raw string literal doesn't process escape sequences, however it does recognize `\\` and `\"` (`\'`) and replaces them with themselves. Thus, a string can have a quote that matches the opening one, but only if it's preceded by a backslash.

```
print("\tchar=\"\\t\"")  # Prints `    char="\t"`.
print(r"\tchar=\"\\t\"") # Prints `\tchar=\"\\t\"`.
```

Note

Some strings cannot be represented using raw string literals: you cannot have an odd number of backslashes at the end of a string or have an unescaped opening quote inside the string. However, in practice this doesn't matter since you can use a different quote type or use concatenation with a regular string literal.

GDScript also supports format strings.

Annotations

Annotations are special tokens in GDScript that act as modifiers to a script or its code and may affect how the script is treated by the Godot engine or editor.

Every annotation starts with the `@` character and is specified by a name. A detailed description and example for each annotation can be found inside the GDScript class reference.

For instance, you can use it to export a value to the editor:

```
@export_range(1, 100, 1, "or_greater")
var ranged_var: int = 50
```

For more information about exporting properties, read the GDScript exports article.

Any constant expression compatible with the required argument type can be passed as an annotation argument value:

```
const MAX_SPEED = 120.0

@export_range(0.0, 0.5 * MAX_SPEED)
var initial_speed: float = 0.25 * MAX_SPEED
```

Annotations can be specified one per line or all in the same line. They affect the next statement that isn't an annotation. Annotations can have arguments sent between parentheses and separated by commas.

Both of these are the same:

```
@annotation_a
@annotation_b
var variable

@annotation_a @annotation_b var variable
```

`@onready` annotation

When using nodes, it's common to desire to keep references to parts of the scene in a variable. As scenes are only warranted to be configured when entering the active scene tree, the sub-nodes can only be obtained when a call to `Node._ready()` is made.

```
var my_label


func _ready():
    my_label = get_node("MyLabel")
```

This can get a little cumbersome, especially when nodes and external references pile up. For this, GDScript has the `@onready` annotation, that defers initialization of a member variable until `_ready()` is called. It can replace the above code with a single line:

```
@onready var my_label = get_node("MyLabel")
```

Warning

Applying `@onready` and any `@export` annotation to the same variable doesn't work as you might expect. The `@onready` annotation will cause the default value to be set after the `@export` takes effect and will override it:

```
@export var a = "init_value_a"
@onready @export var b = "init_value_b"

func _init():
    prints(a, b) # init_value_a <null>

func _notification(what):
    if what == NOTIFICATION_SCENE_INSTANTIATED:
        prints(a, b) # exported_value_a exported_value_b

func _ready():
    prints(a, b) # exported_value_a init_value_b
```

Therefore, the `ONREADY_WITH_EXPORT` warning is generated, which is treated as an error by default. We do not recommend disabling or ignoring it.

Comments

Anything from a `#` to the end of the line is ignored and is considered a comment.

```
# This is a comment.
```

Tip

In the Godot script editor, special keywords are highlighted within comments to bring the user's attention to specific comments:

- **Critical** *(appears in red)*: `ALERT`, `ATTENTION`, `CAUTION`, `CRITICAL`, `DANGER`, `SECURITY`
- **Warning** *(appears in yellow)*: `BUG`, `DEPRECATED`, `FIXME`, `HACK`, `TASK`, `TBD`, `TODO`, `WARNING`
- **Notice** *(appears in green)*: `INFO`, `NOTE`, `NOTICE`, `TEST`, `TESTING`

These keywords are case-sensitive, so they must be written in uppercase for them to be recognized:

```
# In the example below, "TODO" will appear in yellow by
default.
# The `:` symbol after the keyword is not required, but it's
often used.

# TODO: Add more items for the player to choose from.
```

The list of highlighted keywords and their colors can be changed in the **Text Editor > Theme > Comment Markers** section of the Editor Settings.

Code regions

Code regions are special types of comments that the script editor understands as *foldable regions*. This means that after writing code region comments, you can collapse and expand the region by clicking the arrow that appears at the left of the comment. This arrow appears within a purple square to be distinguishable from standard code folding.

The syntax is as follows:

```
# Important: There must be *no* space between the `#` and
`region` or `endregion`.

# Region without a description:
#region
...
#endregion

# Region with a description:
#region Some description that is displayed even when collapsed
...
#endregion
```

Tip

To create a code region quickly, select several lines in the script editor, right-click the selection then choose **Create Code Region**. The region description will be selected automatically for editing.

It is possible to nest code regions within other code regions.

Here's a concrete usage example of code regions:

```
# This comment is outside the code region. It will be visible
when collapsed.
#region Terrain generation
# This comment is inside the code region. It won't be visible
```

```
when collapsed.
func generate_lakes():
    pass

func generate_hills():
    pass
#endregion

#region Terrain population
func place_vegetation():
    pass

func place_roads():
    pass
#endregion
```

This can be useful to organize large chunks of code into easier to understand sections. However, remember that external editors generally don't support this feature, so make sure your code is easy to follow even when not relying on folding code regions.

Note

Individual functions and indented sections (such as `if` and `for`) can *always* be collapsed in the script editor. This means you should avoid using a code region to contain a single function or indented section, as it won't bring much of a benefit. Code regions work best when they're used to group multiple elements together.

Line continuation

A line of code in GDScript can be continued on the next line by using a backslash (`\`). Add one at the end of a line and the code on the next line will act like it's where the backslash is. Here is an example:

```
var a = 1 + \
2
```

A line can be continued multiple times like this:

```
var a = 1 + \
4 + \
10 + \
4
```

Built-in types

Built-in types are stack-allocated. They are passed as values. This means a copy is created on each assignment or when passing them as arguments to functions. The exceptions are `Object`, `Array`, `Dictionary`, and packed arrays (such as `PackedByteArray`), which are passed by reference so they are shared. All arrays, `Dictionary`, and some objects (`Node`, `Resource`) have a `duplicate()` method that allows you to make a copy.

Basic built-in types

A variable in GDScript can be assigned to several built-in types.

null

`null` is an empty data type that contains no information and can not be assigned any other value.

bool

Short for "boolean", it can only contain `true` or `false`.

int

Short for "integer", it stores whole numbers (positive and negative). It is stored as a 64-bit value, equivalent to `int64_t` in C++.

float

Stores real numbers, including decimals, using floating-point values. It is stored as a 64-bit value, equivalent to `double` in C++. Note: Currently, data structures such as `Vector2`, `Vector3`, and `PackedFloat32Array` store 32-bit single-precision `float` values.

String

A sequence of characters in Unicode format [https://en.wikipedia.org/wiki/Unicode].

StringName

An immutable string that allows only one instance of each name. They are slower to create and may result in waiting for locks when multithreading. In exchange, they're very fast to compare, which makes them good candidates for dictionary keys.

NodePath

A pre-parsed path to a node or a node property. It can be easily assigned to, and from, a String. They are useful to interact with the tree to get a node, or affecting properties like with Tweens.

Vector built-in types

Vector2

2D vector type containing `x` and `y` fields. Can also be accessed as an array.

Vector2i

Same as a Vector2 but the components are integers. Useful for representing items in a 2D grid.

Rect2

2D Rectangle type containing two vectors fields: `position` and `size`. Also contains an `end` field which is `position + size`.

Vector3

3D vector type containing `x`, `y` and `z` fields. This can also be accessed as an array.

Vector3i

Same as Vector3 but the components are integers. Can be use for indexing items in a 3D grid.

Transform2D

3×2 matrix used for 2D transforms.

Plane

3D Plane type in normalized form that contains a `normal` vector field and a `d` scalar distance.

Quaternion

Quaternion is a datatype used for representing a 3D rotation. It's useful for interpolating rotations.

AABB

Axis-aligned bounding box (or 3D box) contains 2 vectors fields: `position` and `size`. Also contains an `end` field which is `position + size`.

Basis

3x3 matrix used for 3D rotation and scale. It contains 3 vector fields (`x`, `y` and `z`) and can also be accessed as an array of 3D vectors.

Transform3D

3D Transform contains a Basis field `basis` and a Vector3 field `origin`.

Engine built-in types

Color

Color data type contains `r`, `g`, `b`, and `a` fields. It can also be accessed as `h`, `s`, and `v` for hue/saturation/value.

RID

Resource ID (RID). Servers use generic RIDs to reference opaque data.

Object

Base class for anything that is not a built-in type.

Container built-in types

Array

Generic sequence of arbitrary object types, including other arrays or dictionaries (see below). The array can resize dynamically. Arrays are indexed starting from index `0`. Negative indices count from the end.

```
var arr = []
arr = [1, 2, 3]
var b = arr[1] # This is 2.
var c = arr[arr.size() - 1] # This is 3.
var d = arr[-1] # Same as the previous line, but shorter.
arr[0] = "Hi!" # Replacing value 1 with "Hi!".
arr.append(4) # Array is now ["Hi!", 2, 3, 4].
```

Typed arrays

Godot 4.0 added support for typed arrays. On write operations, Godot checks that element values match the specified type, so the array cannot contain invalid values. The GDScript static analyzer takes typed arrays into account, however array methods like `front()` and `back()` still have the `Variant` return type.

Typed arrays have the syntax `Array[Type]`, where `Type` can be any `Variant` type, native or user class, or enum. Nested array types (like `Array[Array[int]]`) are not supported.

```
var a: Array[int]
var b: Array[Node]
var c: Array[MyClass]
var d: Array[MyEnum]
var e: Array[Variant]
```

`Array` and `Array[Variant]` are the same thing.

Note

Arrays are passed by reference, so the array element type is also an attribute of the in-memory structure referenced by a variable in runtime. The static type of a variable restricts the structures that it can reference to. Therefore, you **cannot** assign an array with a different element type, even if the type is a subtype of the required type.

If you want to *convert* a typed array, you can create a new array and use the Array.assign() method:

```
var a: Array[Node2D] = [Node2D.new()]

# (OK) You can add the value to the array because `Node2D`
extends `Node`.
var b: Array[Node] = [a[0]]

# (Error) You cannot assign an `Array[Node2D]` to an
`Array[Node]` variable.
b = a

# (OK) But you can use the `assign()` method instead. Unlike
the `=` operator,
# the `assign()` method copies the contents of the array, not
the reference.
b.assign(a)
```

The only exception was made for the `Array` (`Array[Variant]`) type, for user convenience and compatibility with old code. However, operations on untyped arrays are considered unsafe.

Packed arrays

GDScript arrays are allocated linearly in memory for speed. Large arrays (more than tens of thousands of elements) may however cause memory fragmentation. If this is a concern, special types of arrays are available. These only accept a single data type. They avoid memory fragmentation and use less memory, but are atomic and tend to run slower than generic arrays. They are therefore only recommended to use for large data sets:

- PackedByteArray: An array of bytes (integers from 0 to 255).
- PackedInt32Array: An array of 32-bit integers.
- PackedInt64Array: An array of 64-bit integers.
- PackedFloat32Array: An array of 32-bit floats.
- PackedFloat64Array: An array of 64-bit floats.
- PackedStringArray: An array of strings.
- PackedVector2Array: An array of Vector2 values.
- PackedVector3Array: An array of Vector3 values.
- PackedVector4Array: An array of Vector4 values.
- PackedColorArray: An array of Color values.

Dictionary

Associative container which contains values referenced by unique keys.

```
var d = {4: 5, "A key": "A value", 28: [1, 2, 3]}
d["Hi!"] = 0
d = {
    22: "value",
    "some_key": 2,
    "other_key": [2, 3, 4],
    "more_key": "Hello"
}
```

Lua-style table syntax is also supported. Lua-style uses `=` instead of `:` and doesn't use quotes to mark string keys (making for slightly less to write). However, keys written in this form can't start with a digit (like any GDScript identifier), and must be string literals.

```
var d = {
    test22 = "value",
    some_key = 2,
    other_key = [2, 3, 4],
    more_key = "Hello"
}
```

To add a key to an existing dictionary, access it like an existing key and assign to it:

```
var d = {} # Create an empty Dictionary.
d.waiting = 14 # Add String "waiting" as a key and assign the value 14 to it.
d[4] = "hello" # Add integer 4 as a key and assign the String "hello" as its value.
d["Godot"] = 3.01 # Add String "Godot" as a key and assign the value 3.01 to it.

var test = 4
# Prints "hello" by indexing the dictionary with a dynamic key.
# This is not the same as `d.test`. The bracket syntax equivalent to
# `d.test` is `d["test"]`.
print(d[test])
```

Note

The bracket syntax can be used to access properties of any Object, not just Dictionaries. Keep in mind it will cause a script error when attempting to index a non-existing property. To avoid this, use the Object.get() and Object.set() methods instead.

Signal

A signal is a message that can be emitted by an object to those who want to listen to it. The Signal type can be used for passing the emitter around.

Signals are better used by getting them from actual objects, e.g. `$Button.button_up`.

Callable

Contains an object and a function, which is useful for passing functions as values (e.g. when connecting to signals).

Getting a method as a member returns a callable. `var x = $Sprite2D.rotate` will set the value of `x` to a callable with `$Sprite2D` as the object and `rotate` as the method.

You can call it using the `call` method: `x.call(PI)`.

Data

Variables

Variables can exist as class members or local to functions. They are created with the `var` keyword and may, optionally, be assigned a value upon initialization.

```
var a # Data type is 'null' by default.
var b = 5
var c = 3.8
var d = b + c # Variables are always initialized in direct order (see below).
```

Variables can optionally have a type specification. When a type is specified, the variable will be forced to have always that same type, and trying to assign an incompatible value will raise an error.

Types are specified in the variable declaration using a `:` (colon) symbol after the variable name, followed by the type.

```
var my_vector2: Vector2
var my_node: Node = Sprite2D.new()
```

If the variable is initialized within the declaration, the type can be inferred, so it's possible to omit the type name:

```
var my_vector2 := Vector2() # 'my_vector2' is of type
'Vector2'.
var my_node := Sprite2D.new() # 'my_node' is of type
'Sprite2D'.
```

Type inference is only possible if the assigned value has a defined type, otherwise it will raise an error.

Valid types are:

- Built-in types (Array, Vector2, int, String, etc.).
- Engine classes (Node, Resource, Reference, etc.).
- Constant names if they contain a script resource (`MyScript` if you declared `const MyScript = preload("res://my_script.gd")`).
- Other classes in the same script, respecting scope (`InnerClass.NestedClass` if you declared `class NestedClass` inside the `class InnerClass` in the same scope).
- Script classes declared with the `class_name` keyword.
- Autoloads registered as singletons.

Note

While `Variant` is a valid type specification, it's not an actual type. It only means there's no set type and is equivalent to not having a static type at all. Therefore, inference is not allowed by default for `Variant`, since it's likely a mistake.

You can turn off this check, or make it only a warning, by changing it in the project settings. See GDScript warning system for details.

Initialization order

Member variables are initialized in the following order:

1. Depending on the variable's static type, the variable is either `null` (untyped variables and objects) or has a default value of the type (`0` for `int`, `false` for `bool`, etc.).
2. The specified values are assigned in the order of the variables in the script, from top to bottom.
    - (Only for `Node`-derived classes) If the `@onready` annotation is applied to a variable, its initialization is deferred to step 5.
3. If defined, the `_init()` method is called.
4. When instantiating scenes and resources, the exported values are assigned.
5. (Only for `Node`-derived classes) `@onready` variables are initialized.
6. (Only for `Node`-derived classes) If defined, the `_ready()` method is called.

Warning

You can specify a complex expression as a variable initializer, including function calls. Make sure the variables are initialized in the correct order, otherwise your values may be overwritten. For example:

```
var a: int = proxy("a", 1)
var b: int = proxy("b", 2)
var _data: Dictionary = {}

func proxy(key: String, value: int):
    _data[key] = value
    print(_data)
    return value

func _init() -> void:
    print(_data)
```

Will print:

```
{ "a": 1 }
{ "a": 1, "b": 2 }
{   }
```

To fix this, move the `_data` variable definition above the `a` definition or remove the empty dictionary assignment (`= {}`).

Static variables

A class member variable can be declared static:

```
static var a
```

Static variables belong to the class, not instances. This means that static variables share values between multiple instances, unlike regular member variables.

From inside a class, you can access static variables from any function, both static and non-static. From outside the class, you can access static variables using the class or an instance (the second is not recommended as it is less readable).

Note

The `@export` and `@onready` annotations cannot be applied to a static variable. Local variables cannot be static.

The following example defines a `Person` class with a static variable named `max_id`. We increment the `max_id` in the `_init()` function. This makes it easy to keep track of the number of `Person` instances in our game.

```
# person.gd
class_name Person

static var max_id = 0
```

```
var id
var name

func _init(p_name):
    max_id += 1
    id = max_id
    name = p_name
```

In this code, we create two instances of our `Person` class and check that the class and every instance have the same `max_id` value, because the variable is static and accessible to every instance.

```
# test.gd
extends Node

func _ready():
    var person1 = Person.new("John Doe")
    var person2 = Person.new("Jane Doe")

    print(person1.id) # 1
    print(person2.id) # 2

    print(Person.max_id)  # 2
    print(person1.max_id) # 2
    print(person2.max_id) # 2
```

Static variables can have type hints, setters and getters:

```
static var balance: int = 0

static var debt: int:
    get:
        return -balance
    set(value):
        balance = -value
```

A base class static variable can also be accessed via a child class:

```
class A:
    static var x = 1
```

```
class B extends A:
    pass

func _ready():
    prints(A.x, B.x) # 1 1
    A.x = 2
    prints(A.x, B.x) # 2 2
    B.x = 3
    prints(A.x, B.x) # 3 3
```

`@static_unload` annotation

Since GDScript classes are resources, having static variables in a script prevents it from being unloaded even if there are no more instances of that class and no other references left. This can be important if static variables store large amounts of data or hold references to other project resources, such as scenes. You should clean up this data manually, or use the @static_unload annotation if static variables don't store important data and can be reset.

Warning

Currently, due to a bug, scripts are never freed, even if `@static_unload` annotation is used.

Note that `@static_unload` applies to the entire script (including inner classes) and must be placed at the top of the script, before `class_name` and `extends`:

```
@static_unload
class_name MyNode
extends Node
```

See also Static functions and Static constructor.

Casting

Values assigned to typed variables must have a compatible type. If it's needed to coerce a value to be of a certain type, in particular for object types, you can use the casting operator `as`.

Casting between object types results in the same object if the value is of the same type or a subtype of the cast type.

```
var my_node2D: Node2D
my_node2D = $Sprite2D as Node2D # Works since Sprite2D is a
subtype of Node2D.
```

If the value is not a subtype, the casting operation will result in a `null` value.

```
var my_node2D: Node2D
my_node2D = $Button as Node2D # Results in 'null' since a
Button is not a subtype of Node2D.
```

For built-in types, they will be forcibly converted if possible, otherwise the engine will raise an error.

```
var my_int: int
my_int = "123" as int # The string can be converted to int.
my_int = Vector2() as int # A Vector2 can't be converted to
int, this will cause an error.
```

Casting is also useful to have better type-safe variables when interacting with the scene tree:

```
# Will infer the variable to be of type Sprite2D.
var my_sprite := $Character as Sprite2D

# Will fail if $AnimPlayer is not an AnimationPlayer, even if
it has the method 'play()'.
($AnimPlayer as AnimationPlayer).play("walk")
```

Constants

Constants are values you cannot change when the game is running. Their value must be known at compile-time. Using the `const` keyword allows you to give a constant value a name. Trying to assign a value to a constant after it's declared will give you an error.

We recommend using constants whenever a value is not meant to change.

```
const A = 5
const B = Vector2(20, 20)
const C = 10 + 20 # Constant expression.
const D = Vector2(20, 30).x # Constant expression: 20.
const E = [1, 2, 3, 4][0] # Constant expression: 1.
const F = sin(20) # 'sin()' can be used in constant
expressions.
const G = x + 20 # Invalid; this is not a constant expression!
const H = A + 20 # Constant expression: 25 (`A` is a constant).
```

Although the type of constants is inferred from the assigned value, it's also possible to add explicit type specification:

```
const A: int = 5
const B: Vector2 = Vector2()
```

Assigning a value of an incompatible type will raise an error.

You can also create constants inside a function, which is useful to name local magic values.

Enums

Enums are basically a shorthand for constants, and are pretty useful if you want to assign consecutive integers to some constant.

```
enum {TILE_BRICK, TILE_FLOOR, TILE_SPIKE, TILE_TELEPORT}

# Is the same as:
const TILE_BRICK = 0
const TILE_FLOOR = 1
const TILE_SPIKE = 2
const TILE_TELEPORT = 3
```

If you pass a name to the enum, it will put all the keys inside a constant [Dictionary](#) of that name. This means all constant methods of a dictionary can also be used with a named enum.

Important

Keys in a named enum are not registered as global constants. They should be accessed prefixed by the enum's name (`Name.KEY`).

```
enum State {STATE_IDLE, STATE_JUMP = 5, STATE_SHOOT}

# Is the same as:
const State = {STATE_IDLE = 0, STATE_JUMP = 5, STATE_SHOOT = 6}
# Access values with State.STATE_IDLE, etc.

func _ready():
    # Access values with Name.KEY, prints '5'
    print(State.STATE_JUMP)
    # Use dictionary methods:
    # prints '["STATE_IDLE", "STATE_JUMP", "STATE_SHOOT"]'
    print(State.keys())
    # prints '{ "STATE_IDLE": 0, "STATE_JUMP": 5,
"STATE_SHOOT": 6 }'
    print(State)
    # prints '[0, 5, 6]'
    print(State.values())
```

Functions

Functions always belong to a [class](#). The scope priority for variable look-up is: local → class member → global. The `self` variable is always available and is provided as an option for accessing class members, but is not always required (and should *not* be sent as the function's first argument, unlike Python).

```
func my_function(a, b):
    print(a)
    print(b)
    return a + b  # Return is optional; without it 'null' is returned.
```

A function can `return` at any point. The default return value is `null`.

If a function contains only one line of code, it can be written on one line:

```
func square(a): return a * a

func hello_world(): print("Hello World")

func empty_function(): pass
```

Functions can also have type specification for the arguments and for the return value. Types for arguments can be added in a similar way to variables:

```
func my_function(a: int, b: String):
    pass
```

If a function argument has a default value, it's possible to infer the type:

```
func my_function(int_arg := 42, String_arg := "string"):
    pass
```

The return type of the function can be specified after the arguments list using the arrow token (`->`):

```
func my_int_function() -> int:
    return 0
```

Functions that have a return type **must** return a proper value. Setting the type as `void` means the function doesn't return anything. Void functions can return early with the `return` keyword, but they can't return any value.

```
func void_function() -> void:
    return # Can't return a value.
```

Note

Non-void functions must **always** return a value, so if your code has branching statements (such as an `if`/`else` construct), all the possible paths must have a return. E.g., if you have a `return` inside an `if` block but not after it, the editor will raise an error because if the block is not executed, the function won't have a valid value to return.

Referencing functions

Functions are first-class values in terms of the Callable object. Referencing a function by name without calling it will automatically generate the proper callable. This can be used to pass functions as arguments.

```
func map(arr: Array, function: Callable) -> Array:
    var result = []
    for item in arr:
        result.push_back(function.call(item))
    return result

func add1(value: int) -> int:
    return value + 1;

func _ready() -> void:
    var my_array = [1, 2, 3]
    var plus_one = map(my_array, add1)
    print(plus_one) # Prints `[2, 3, 4]`.
```

Note

Callables **must** be called with the call() method. You cannot use the `()` operator directly. This behavior is implemented to avoid performance issues on direct function calls.

Lambda functions

Lambda functions allow you to declare functions that do not belong to a class. Instead, a Callable object is created and assigned to a variable directly. This can be useful to create callables to pass around without polluting the class scope.

```
var lambda = func (x):
    print(x)
```

To call the created lambda you can use the call() method:

```
lambda.call(42) # Prints `42`.
```

Lambda functions can be named for debugging purposes (the name is displayed in the Debugger):

```
var lambda = func my_lambda(x):
    print(x)
```

You can specify type hints for lambda functions in the same way as for regular ones:

```
var lambda := func (x: int) -> void:
    print(x)
```

Note that if you want to return a value from a lambda function, an explicit `return` is required (you can't omit `return`):

```
var lambda = func (x): return x ** 2
print(lambda.call(2)) # Prints `4`.
```

Lambda functions capture the local environment:

```
var x = 42
var lambda = func ():
    print(x) # Prints `42`.
lambda.call()
```

Warning

Local variables are captured by value once, when the lambda is created. So they won't be updated in the lambda if reassigned in the outer function:

```
var x = 42
var lambda = func (): print(x)
lambda.call() # Prints `42`.
x = "Hello"
lambda.call() # Prints `42`.
```

Also, a lambda cannot reassign an outer local variable. After exiting the lambda, the variable will be unchanged, because the lambda capture implicitly shadows it:

```
var x = 42
var lambda = func ():
    print(x) # Prints `42`.
    x = "Hello" # Produces the
`CONFUSABLE_CAPTURE_REASSIGNMENT` warning.
    print(x) # Prints `Hello`.
lambda.call()
print(x) # Prints `42`.
```

However, if you use pass-by-reference data types (arrays, dictionaries, and objects), then the content changes are shared until you reassign the variable:

```
var a = []
var lambda = func ():
    a.append(1)
    print(a) # Prints `[1]`.
    a = [2] # Produces the `CONFUSABLE_CAPTURE_REASSIGNMENT`
warning.
    print(a) # Prints `[2]`.
```

```
lambda.call()
print(a) # Prints `[1]`.
```

Static functions

A function can be declared static. When a function is static, it has no access to the instance member variables or `self`. A static function has access to static variables. Also static functions are useful to make libraries of helper functions:

```
static func sum2(a, b):
    return a + b
```

Lambda functions cannot be declared static.

See also Static variables and Static constructor.

Statements and control flow

Statements are standard and can be assignments, function calls, control flow structures, etc (see below). `;` as a statement separator is entirely optional.

Expressions

Expressions are sequences of operators and their operands in orderly fashion. An expression by itself can be a statement too, though only calls are reasonable to use as statements since other expressions don't have side effects.

Expressions return values that can be assigned to valid targets. Operands to some operator can be another expression. An assignment is not an expression and thus does not return any value.

Here are some examples of expressions:

```
2 + 2 # Binary operation.
-5 # Unary operation.
"okay" if x > 4 else "not okay" # Ternary operation.
x # Identifier representing variable or constant.
x.a # Attribute access.
x[4] # Subscript access.
x > 2 or x < 5 # Comparisons and logic operators.
x == y + 2 # Equality test.
do_something() # Function call.
[1, 2, 3] # Array definition.
{A = 1, B = 2} # Dictionary definition.
preload("res://icon.png") # Preload builtin function.
self # Reference to current instance.
```

Identifiers, attributes, and subscripts are valid assignment targets. Other expressions cannot be on the left side of an assignment.

if/else/elif

Simple conditions are created by using the `if`/`else`/`elif` syntax. Parenthesis around conditions are allowed, but not required. Given the nature of the tab-based indentation, `elif` can be used instead of `else`/`if` to maintain a level of indentation.

```
if (expression):
    statement(s)
elif (expression):
    statement(s)
else:
    statement(s)
```

Short statements can be written on the same line as the condition:

```
if 1 + 1 == 2: return 2 + 2
else:
    var x = 3 + 3
    return x
```

Sometimes, you might want to assign a different initial value based on a boolean expression. In this case, ternary-if expressions come in handy:

```
var x = (value) if (expression) else (value)
y += 3 if y < 10 else -1
```

Ternary-if expressions can be nested to handle more than 2 cases. When nesting ternary-if expressions, it is recommended to wrap the complete expression over multiple lines to preserve readability:

```
var count = 0

var fruit = (
        "apple" if count == 2
        else "pear" if count == 1
        else "banana" if count == 0
        else "orange"
)
print(fruit)  # banana

# Alternative syntax with backslashes instead of parentheses
(for multi-line expressions).
# Less lines required, but harder to refactor.
var fruit_alt = \
        "apple" if count == 2 \
        else "pear" if count == 1 \
        else "banana" if count == 0 \
        else "orange"
print(fruit_alt)  # banana
```

You may also wish to check if a value is contained within something. You can use an `if` statement combined with the `in` operator to accomplish this:

```
# Check if a letter is in a string.
var text = "abc"
if 'b' in text: print("The string contains b")

# Check if a variable is contained within a node.
if "varName" in get_parent(): print("varName is defined in
parent!")
```

while

Simple loops are created by using `while` syntax. Loops can be broken using `break` or continued using `continue` (which skips to the next iteration of the loop without executing any further code in the current iteration):

```
while (expression):
    statement(s)
```

for

To iterate through a range, such as an array or table, a *for* loop is used. When iterating over an array, the current array element is stored in the loop variable. When iterating over a dictionary, the *key* is stored in the loop variable.

```
for x in [5, 7, 11]:
    statement # Loop iterates 3 times with 'x' as 5, then 7 and
finally 11.

var names = ["John", "Marta", "Samantha", "Jimmy"]
for name: String in names: # Typed loop variable.
    print(name) # Prints name's content.

var dict = {"a": 0, "b": 1, "c": 2}
for i in dict:
    print(dict[i]) # Prints 0, then 1, then 2.

for i in range(3):
    statement # Similar to [0, 1, 2] but does not allocate an
array.

for i in range(1, 3):
    statement # Similar to [1, 2] but does not allocate an
array.

for i in range(2, 8, 2):
    statement # Similar to [2, 4, 6] but does not allocate an
array.
```

```
for i in range(8, 2, -2):
    statement # Similar to [8, 6, 4] but does not allocate an
array.

for c in "Hello":
    print(c) # Iterate through all characters in a String,
print every letter on new line.

for i in 3:
    statement # Similar to range(3).

for i in 2.2:
    statement # Similar to range(ceil(2.2)).
```

If you want to assign values on an array as it is being iterated through, it is best to use `for i in array.size()`.

```
for i in array.size():
    array[i] = "Hello World"
```

The loop variable is local to the for-loop and assigning to it will not change the value on the array. Objects passed by reference (such as nodes) can still be manipulated by calling methods on the loop variable.

```
for string in string_array:
    string = "Hello World" # This has no effect

for node in node_array:
    node.add_to_group("Cool_Group") # This has an effect
```

match

A `match` statement is used to branch execution of a program. It's the equivalent of the `switch` statement found in many other languages, but offers some additional features.

Warning

`match` is more type strict than the `==` operator. For example `1` will **not** match `1.0`. The only exception is `String` vs `StringName` matching: for example, the String `"hello"` is considered equal to the StringName `&"hello"`.

Basic syntax

```
match <test value>:
    <pattern(s)>:
        <block>
    <pattern(s)> when <pattern guard>:
        <block>
    <...>
```

Crash-course for people who are familiar with switch statements

1. Replace `switch` with `match`.
2. Remove `case`.
3. Remove any `break`s.
4. Change `default` to a single underscore.

Control flow

The patterns are matched from top to bottom. If a pattern matches, the first corresponding block will be executed. After that, the execution continues below the `match` statement.

Note

The special `continue` behavior in `match` supported in 3.x was removed in Godot 4.0.

The following pattern types are available:

- Literal pattern

  Matches a literal:

  ```
  match x:
      1:
          print("We are number one!")
      2:
          print("Two are better than one!")
      "test":
          print("Oh snap! It's a string!")
  ```

- Expression pattern

  Matches a constant expression, an identifier, or an attribute access (`A.B`):

  ```
  match typeof(x):
      TYPE_FLOAT:
          print("float")
      TYPE_STRING:
          print("text")
      TYPE_ARRAY:
          print("array")
  ```

- Wildcard pattern

  This pattern matches everything. It's written as a single underscore.

  It can be used as the equivalent of the `default` in a `switch` statement in other languages:

  ```
  match x:
      1:
          print("It's one!")
      2:
          print("It's one times two!")
      _:
          print("It's not 1 or 2. I don't care to be
  honest.")
  ```

- Binding pattern

  A binding pattern introduces a new variable. Like the wildcard pattern, it matches everything - and also gives that value a name. It's especially useful in array and dictionary patterns:

  ```
  match x:
      1:
          print("It's one!")
      2:
          print("It's one times two!")
      var new_var:
          print("It's not 1 or 2, it's ", new_var)
  ```

- Array pattern

  Matches an array. Every single element of the array pattern is a pattern itself, so you can nest them.

  The length of the array is tested first, it has to be the same size as the pattern, otherwise the pattern doesn't match.

  Open-ended array: An array can be bigger than the pattern by making the last subpattern `..`

  Every subpattern has to be comma-separated.

  ```
  match x:
      []:
          print("Empty array")
      [1, 3, "test", null]:
          print("Very specific array")
      [var start, _, "test"]:
          print("First element is ", start, ", and the
  last is \"test\"")
      [42, ..]:
          print("Open ended array")
  ```

- Dictionary pattern

Works in the same way as the array pattern. Every key has to be a constant pattern.

The size of the dictionary is tested first, it has to be the same size as the pattern, otherwise the pattern doesn't match.

Open-ended dictionary: A dictionary can be bigger than the pattern by making the last subpattern `..`.

Every subpattern has to be comma separated.

If you don't specify a value, then only the existence of the key is checked.

A value pattern is separated from the key pattern with a `:`.

```
match x:
    {}:
        print("Empty dict")
    {"name": "Dennis"}:
        print("The name is Dennis")
    {"name": "Dennis", "age": var age}:
        print("Dennis is ", age, " years old.")
    {"name", "age"}:
        print("Has a name and an age, but it's not Dennis :(")
    {"key": "godotisawesome", ..}:
        print("I only checked for one entry and ignored the rest")
```

- Multiple patterns

  You can also specify multiple patterns separated by a comma. These patterns aren't allowed to have any bindings in them.

```
match x:
    1, 2, 3:
        print("It's 1 - 3")
    "Sword", "Splash potion", "Fist":
        print("Yep, you've taken damage")
```

Pattern guards

A *pattern guard* is an optional condition that follows the pattern list and allows you to make additional checks before choosing a `match` branch. Unlike a pattern, a pattern guard can be an arbitrary expression.

Only one branch can be executed per `match`. Once a branch is chosen, the rest are not checked. If you want to use the same pattern for multiple branches or to prevent choosing a branch with too general pattern, you can specify a pattern guard after the list of patterns with the `when` keyword:

```
match point:
    [0, 0]:
        print("Origin")
    [_, 0]:
        print("Point on X-axis")
    [0, _]:
        print("Point on Y-axis")
    [var x, var y] when y == x:
        print("Point on line y = x")
    [var x, var y] when y == -x:
        print("Point on line y = -x")
    [var x, var y]:
        print("Point (%s, %s)" % [x, y])
```

- If there is no matching pattern for the current branch, the pattern guard is **not** evaluated and the patterns of the next branch are checked.
- If a matching pattern is found, the pattern guard is evaluated.
    - If it's true, then the body of the branch is executed and `match` ends.

  - If it's false, then the patterns of the next branch are checked.

Classes

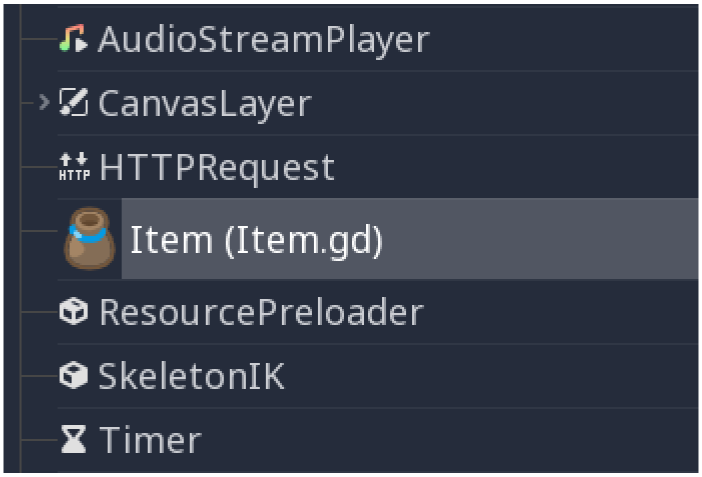
By default, all script files are unnamed classes. In this case, you can only reference them using the file's path, using either a relative or an absolute path. For example, if you name a script file `character.gd`:

```
# Inherit from 'character.gd'.

extends "res://path/to/character.gd"

# Load character.gd and create a new node instance from it.

var Character = load("res://path/to/character.gd")
var character_node = Character.new()
```

Registering named classes

You can give your class a name to register it as a new type in Godot's editor. For that, you use the `class_name` keyword. You can optionally use the `@icon` annotation with a path to an image, to use it as an icon. Your class will then appear with its new icon in the editor:

```
# item.gd

@icon("res://interface/icons/item.png")
class_name Item
extends Node
```


AudioStreamPlayer
CanvasLayer
HTTPRequest
Item (Item.gd)
ResourcePreloader
SkeletonIK
Timer

Tip

SVG images that are used as custom node icons should have the **Editor > Scale With Editor Scale** and **Editor > Convert Icons With Editor Theme** import options enabled. This allows icons to follow the editor's scale and theming settings if the icons are designed with the same color palette as Godot's own icons.

Here's a class file example:

```
# Saved as a file named 'character.gd'.

class_name Character


var health = 5


func print_health():
    print(health)


func print_this_script_three_times():
    print(get_script())
    print(ResourceLoader.load("res://character.gd"))
    print(Character)
```

If you want to use `extends` too, you can keep both on the same line:

```
class_name MyNode extends Node
```

Note

Godot initializes non-static variables every time you create an instance, and this includes arrays and dictionaries. This is in the spirit of thread safety, since scripts can be initialized in separate threads without the user knowing.

Warning

The Godot editor will hide these custom classes with names that beging with the prefix "Editor" in the 'Create New Node' or 'Create New Scene' dialog windows. The classes are available for instantiation at runtime via their class names, but are automatically hidden by the editor windows along with the built-in editor nodes used by the Godot editor.

Inheritance

A class (stored as a file) can inherit from:

- A global class.
- Another class file.
- An inner class inside another class file.

Multiple inheritance is not allowed.

Inheritance uses the `extends` keyword:

```
# Inherit/extend a globally available class.
extends SomeClass
```

```
# Inherit/extend a named class file.
extends "somefile.gd"

# Inherit/extend an inner class in another file.
extends "somefile.gd".SomeInnerClass
```

Note

If inheritance is not explicitly defined, the class will default to inheriting RefCounted.

To check if a given instance inherits from a given class, the `is` keyword can be used:

```
# Cache the enemy class.
const Enemy = preload("enemy.gd")

# [...]

# Use 'is' to check inheritance.
if entity is Enemy:
    entity.apply_damage()
```

To call a function in a *super class* (i.e. one `extend`-ed in your current class), use the `super` keyword:

```
super(args)
```

This is especially useful because functions in extending classes replace functions with the same name in their super classes. If you still want to call them, you can use `super`:

```
func some_func(x):
    super(x) # Calls the same function on the super class.
```

If you need to call a different function from the super class, you can specify the function name with the attribute operator:

```
func overriding():
    return 0 # This overrides the method in the base class.

func dont_override():
    return super.overriding() # This calls the method as
defined in the base class.
```

Warning

One of the common misconceptions is trying to override *non-virtual* engine methods such as `get_class()`, `queue_free()`, etc. This is not supported for technical reasons.

In Godot 3, you can *shadow* engine methods in GDScript, and it will work if you call this method in GDScript. However, the engine will **not** execute your code if the method is called inside the engine on some event.

In Godot 4, even shadowing may not always work, as GDScript optimizes native method calls. Therefore, we added the `NATIVE_METHOD_OVERRIDE` warning, which is treated as an error by default. We strongly advise against disabling or ignoring the warning.

Note that this does not apply to virtual methods such as `_ready()`, `_process()` and others (marked with the `virtual` qualifier in the documentation and the names start with an underscore). These methods are specifically for customizing engine behavior and can be overridden in GDScript. Signals and notifications can also be useful for these purposes.

Class constructor

The class constructor, called on class instantiation, is named `_init`. If you want to call the base class constructor, you can also use the `super` syntax. Note that every class has an implicit constructor that is always called (defining the default values of class variables). `super` is used to call the explicit constructor:

```
func _init(arg):
   super("some_default", arg) # Call the custom base constructor.
```

This is better explained through examples. Consider this scenario:

```
# state.gd (inherited class).
var entity = null
var message = null


func _init(e=null):
    entity = e


func enter(m):
    message = m


# idle.gd (inheriting class).
extends "state.gd"


func _init(e=null, m=null):
    super(e)
    # Do something with 'e'.
    message = m
```

There are a few things to keep in mind here:

1. If the inherited class (`state.gd`) defines a `_init` constructor that takes arguments (`e` in this case), then the inheriting class (`idle.gd`) *must* define `_init` as well and pass appropriate parameters to `_init` from `state.gd`.

2. `idle.gd` can have a different number of arguments than the base class `state.gd`.

3. In the example above, `e` passed to the `state.gd` constructor is the same `e` passed in to `idle.gd`.

4. If `idle.gd`'s `_init` constructor takes 0 arguments, it still needs to pass some value to the `state.gd` base class, even if it does nothing. This brings us to the fact that you can pass expressions to the base constructor as well, not just variables, e.g.:

   ```
   # idle.gd

   func _init():
       super(5)
   ```

Static constructor

A static constructor is a static function `_static_init` that is called automatically when the class is loaded, after the static variables have been initialized:

```
static var my_static_var = 1

static func _static_init():
    my_static_var = 2
```

A static constructor cannot take arguments and must not return any value.

Inner classes

A class file can contain inner classes. Inner classes are defined using the `class` keyword. They are instanced using the `ClassName.new()` function.

```
# Inside a class file.

# An inner class in this class file.
class SomeInnerClass:
    var a = 5
```

```
    func print_value_of_a():
        print(a)


# This is the constructor of the class file's main class.
func _init():
    var c = SomeInnerClass.new()
    c.print_value_of_a()
```

Classes as resources

Classes stored as files are treated as GDScripts. They must be loaded from disk to access them in other classes. This is done using either the `load` or `preload` functions (see below). Instancing of a loaded class resource is done by calling the `new` function on the class object:

```
# Load the class resource when calling load().
var MyClass = load("myclass.gd")

# Preload the class only once at compile time.
const MyClass = preload("myclass.gd")


func _init():
    var a = MyClass.new()
    a.some_function()
```

Exports

Note

Documentation about exports has been moved to GDScript exported properties.

Properties (setters and getters)

Sometimes, you want a class' member variable to do more than just hold data and actually perform some validation or computation whenever its value changes. It may also be desired to encapsulate its access in some way.

For this, GDScript provides a special syntax to define properties using the `set` and `get` keywords after a variable declaration. Then you can define a code block that will be executed when the variable is accessed or assigned.

Example:

```
var milliseconds: int = 0
var seconds: int:
    get:
        return milliseconds / 1000
    set(value):
        milliseconds = value * 1000
```

Note

Unlike `setget` in previous Godot versions, `set` and `get` methods are **always** called (except as noted below), even when accessed inside the same class (with or without prefixing with `self.`). This makes the behavior consistent. If you need direct access to the value, use another variable for direct access and make the property code use that name.

Alternative syntax

Also there is another notation to use existing class functions if you want to split the code from the variable declaration or you need to reuse the code across multiple properties (but you can't distinguish which property the setter/getter is being called for):

```
var my_prop:
    get = get_my_prop, set = set_my_prop
```

This can also be done in the same line:

```
var my_prop: get = get_my_prop, set = set_my_prop
```

The setter and getter must use the same notation, mixing styles for the same variable is not allowed.

Note

You cannot specify type hints for *inline* setters and getters. This is done on purpose to reduce the boilerplate. If the variable is typed, then the setter's argument is automatically of the same type, and the getter's return value must match it. Separated setter/getter functions can have type hints, and the type must match the variable's type or be a wider type.

When setter/getter is not called

When a variable is initialized, the value of the initializer will be written directly to the variable. Including if the `@onready` annotation is applied to the variable.

Using the variable's name to set it inside its own setter or to get it inside its own getter will directly access the underlying member, so it won't generate infinite recursion and saves you from explicitly declaring another variable:

```
signal changed(new_value)
var warns_when_changed = "some value":
    get:
        return warns_when_changed
    set(value):
        changed.emit(value)
        warns_when_changed = value
```

This also applies to the alternative syntax:

```
var my_prop: set = set_my_prop

func set_my_prop(value):
    my_prop = value # No infinite recursion.
```

Warning

The exception does **not** propagate to other functions called in the setter/getter. For example, the following code **will** cause an infinite recursion:

```
var my_prop:
    set(value):
        set_my_prop(value)

func set_my_prop(value):
    my_prop = value # Infinite recursion, since
`set_my_prop()` is not the setter.
```

Tool mode

By default, scripts don't run inside the editor and only the exported properties can be changed. In some cases, it is desired that they do run inside the editor (as long as they don't execute game code or manually avoid doing so). For this, the `@tool` annotation exists and must be placed at the top of the file:

```
@tool
extends Button
```

```
func _ready():
    print("Hello")
```

See Running code in the editor for more information.

Warning

Be cautious when freeing nodes with `queue_free()` or `free()` in a tool script (especially the script's owner itself). As tool scripts run their code in the editor, misusing them may lead to crashing the editor.

Memory management

Godot implements reference counting to free certain instances that are no longer used, instead of a garbage collector, or requiring purely manual management. Any instance of the RefCounted class (or any class that inherits it, such as Resource) will be freed automatically when no longer in use. For an instance of any class that is not a RefCounted (such as Node or the base Object type), it will remain in memory until it is deleted with `free()` (or `queue_free()` for Nodes).

Note

If a Node is deleted via `free()` or `queue_free()`, all of its children will also recursively be deleted.

To avoid reference cycles that can't be freed, a WeakRef function is provided for creating weak references, which allow access to the object without preventing a RefCounted from freeing. Here is an example:

```
extends Node

var my_file_ref

func _ready():
    var f = FileAccess.open("user://example_file.json",
FileAccess.READ)
    my_file_ref = weakref(f)
    # the FileAccess class inherits RefCounted, so it will be
freed when not in use

    # the WeakRef will not prevent f from being freed when
other_node is finished
    other_node.use_file(f)

func _this_is_called_later():
    var my_file = my_file_ref.get_ref()
    if my_file:
        my_file.close()
```

Alternatively, when not using references, the `is_instance_valid(instance)` can be used to check if an object has been freed.

Signals

Signals are a tool to emit messages from an object that other objects can react to. To create custom signals for a class, use the `signal` keyword.

```
extends Node


# A signal named health_depleted.
signal health_depleted
```

Note

Signals are a Callback [https://en.wikipedia.org/wiki/Callback_(computer_programming)] mechanism. They also fill the role of Observers, a common programming pattern. For more information, read the Observer tutorial [https://gameprogrammingpatterns.com/observer.html] in the Game Programming Patterns ebook.

You can connect these signals to methods the same way you connect built-in signals of nodes like Button or RigidBody3D.

In the example below, we connect the `health_depleted` signal from a `Character` node to a `Game` node. When the `Character` node emits the signal, the game node's `_on_character_health_depleted` is called:

```
# game.gd

func _ready():
    var character_node = get_node('Character')

character_node.health_depleted.connect(_on_character_health_dep
leted)


func _on_character_health_depleted():
    get_tree().reload_current_scene()
```

You can emit as many arguments as you want along with a signal.

Here is an example where this is useful. Let's say we want a life bar on screen to react to health changes with an animation, but we want to keep the user interface separate from the player in our scene tree.

In our `character.gd` script, we define a `health_changed` signal and emit it with Signal.emit(), and from a `Game` node higher up our scene tree, we connect it to the `Lifebar` using the Signal.connect() method:

```
# character.gd

...
signal health_changed


func take_damage(amount):
    var old_health = health
    health -= amount

    # We emit the health_changed signal every time the
    # character takes damage.
    health_changed.emit(old_health, health)
...
```

```
# lifebar.gd

# Here, we define a function to use as a callback when the
# character's health_changed signal is emitted.

...
func _on_Character_health_changed(old_value, new_value):
    if old_value > new_value:
        progress_bar.modulate = Color.RED
    else:
        progress_bar.modulate = Color.GREEN

    # Imagine that `animate` is a user-defined function that
animates the
    # bar filling up or emptying itself.
    progress_bar.animate(old_value, new_value)
...
```

In the `Game` node, we get both the `Character` and `Lifebar` nodes, then connect the character, that emits the signal, to the receiver, the `Lifebar` node in this case.

```
# game.gd

func _ready():
    var character_node = get_node('Character')
    var lifebar_node = get_node('UserInterface/Lifebar')
```

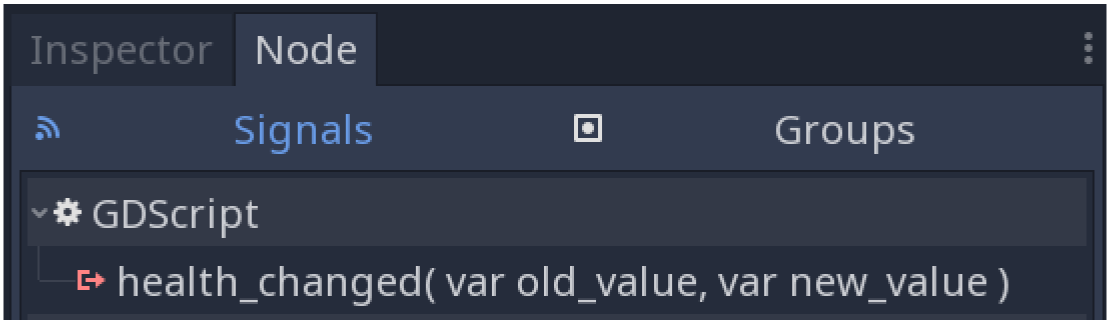
```
character_node.health_changed.connect(lifebar_node._on_Characte
r_health_changed)
```

This allows the `Lifebar` to react to health changes without coupling it to the `Character` node.

You can write optional argument names in parentheses after the signal's definition:

```
# Defining a signal that forwards two arguments.
signal health_changed(old_value, new_value)
```

These arguments show up in the editor's node dock, and Godot can use them to generate callback functions for you. However, you can still emit any number of arguments when you emit signals; it's up to you to emit the correct values.



health_changed(var old_value, var new_value)

GDScript can bind an array of values to connections between a signal and a method. When the signal is emitted, the callback method receives the bound values. These bound arguments are unique to each connection, and the values will stay the same.

You can use this array of values to add extra constant information to the connection if the emitted signal itself doesn't give you access to all the data that you need.

Building on the example above, let's say we want to display a log of the damage taken by each character on the screen, like `Player1 took 22 damage.`. The `health_changed` signal doesn't give us the name of the character that took damage. So when we connect the signal to the in-game console, we can add the character's name in the binds array argument:

```
# game.gd

func _ready():
    var character_node = get_node('Character')
    var battle_log_node = get_node('UserInterface/BattleLog')


character_node.health_changed.connect(battle_log_node._on_Chara
cter_health_changed.bind(character_node.name))
```

Our `BattleLog` node receives each element in the binds array as an extra argument:

```
# battle_log.gd

func _on_Character_health_changed(old_value, new_value,
character_name):
    if not new_value <= old_value:
        return

    var damage = old_value - new_value
    label.text += character_name + " took " + str(damage) + "
damage."
```

Awaiting signals or coroutines

The `await` keyword can be used to create coroutines [https://en.wikipedia.org/wiki/Coroutine] which wait until a signal is emitted before continuing execution. Using the `await` keyword with a signal or a call to a function that is also a coroutine will immediately return the control to the caller. When the signal is emitted (or the called coroutine finishes), it will resume execution from the point on where it stopped.

For example, to stop execution until the user presses a button, you can do something like this:

```
func wait_confirmation():
    print("Prompting user")
    await $Button.button_up # Waits for the button_up signal
from Button node.
    print("User confirmed")
    return true
```

In this case, the `wait_confirmation` becomes a coroutine, which means that the caller also needs to await it:

```
func request_confirmation():
    print("Will ask the user")
    var confirmed = await wait_confirmation()
    if confirmed:
        print("User confirmed")
    else:
        print("User cancelled")
```

Note that requesting a coroutine's return value without `await` will trigger an error:

```
func wrong():
    var confirmed = wait_confirmation() # Will give an error.
```

However, if you don't depend on the result, you can just call it asynchronously, which won't stop execution and won't make the current function a coroutine:

```
func okay():
    wait_confirmation()
    print("This will be printed immediately, before the user
press the button.")
```

If you use await with an expression that isn't a signal nor a coroutine, the value will be returned immediately and the function won't give the control back to the caller:

```
func no_wait():
    var x = await get_five()
    print("This doesn't make this function a coroutine.")

func get_five():
    return 5
```

This also means that returning a signal from a function that isn't a coroutine will make the caller await that signal:

```
func get_signal():
    return $Button.button_up

func wait_button():
    await get_signal()
    print("Button was pressed")
```

Note

Unlike `yield` in previous Godot versions, you cannot obtain the function state object. This is done to ensure type safety. With this type safety in place, a function cannot say that it returns an `int` while it actually returns a function state object during runtime.

Assert keyword

The `assert` keyword can be used to check conditions in debug builds. These assertions are ignored in non-debug builds. This means that the expression passed as argument won't be evaluated in a project exported in release mode. Due to this, assertions must **not** contain expressions that have side effects. Otherwise, the behavior of the script would vary depending on whether the project is run in a debug build.

```
# Check that 'i' is 0. If 'i' is not 0, an assertion error will occur.
assert(i == 0)
```

When running a project from the editor, the project will be paused if an assertion error occurs.

You can optionally pass a custom error message to be shown if the assertion fails:

```
assert(enemy_power < 256, "Enemy is too powerful!")
```

GDScript: An introduction to dynamic languages

About

This tutorial aims to be a quick reference for how to use GDScript more efficiently. It focuses on common cases specific to the language, but also covers a lot of information on dynamically typed languages.

It's meant to be especially useful for programmers with little or no previous experience with dynamically typed languages.

Dynamic nature

Pros & cons of dynamic typing

GDScript is a Dynamically Typed language. As such, its main advantages are that:

- The language is easy to get started with.
- Most code can be written and changed quickly and without hassle.
- Less code written means less errors & mistakes to fix.
- The code is easy to read (little clutter).
- No compilation is required to test.
- Runtime is tiny.
- It has duck-typing and polymorphism by nature.

While the main disadvantages are:

- Less performance than statically typed languages.
- More difficult to refactor (symbols can't be traced).
- Some errors that would typically be detected at compile time in statically typed languages only appear while running the code (because expression parsing is more strict).
- Less flexibility for code-completion (some variable types are only known at run-time).

This, translated to reality, means that Godot used with GDScript is a combination designed to create games quickly and efficiently. For games that are very computationally intensive and can't benefit from the engine built-in tools (such as the Vector types, Physics Engine, Math library, etc), the possibility of using C++ is present too. This allows you to still create most of the game in GDScript and add small bits of C++ in the areas that need a performance boost.

Variables & assignment

All variables in a dynamically typed language are "variant"-like. This means that their type is not fixed, and is only modified through assignment. Example:

Static:

```
int a; // Value uninitialized.
a = 5; // This is valid.
a = "Hi!"; // This is invalid.
```

Dynamic:

```
var a # 'null' by default.
a = 5 # Valid, 'a' becomes an integer.
a = "Hi!" # Valid, 'a' changed to a string.
```

As function arguments:

Functions are of dynamic nature too, which means they can be called with different arguments, for example:

Static:

```
void print_value(int value) {

    printf("value is %i\n", value);
}

[..]

print_value(55); // Valid.
print_value("Hello"); // Invalid.
```

Dynamic:

```
func print_value(value):
    print(value)

[..]

print_value(55) # Valid.
print_value("Hello") # Valid.
```

Pointers & referencing:

In static languages, such as C or C++ (and to some extent Java and C#), there is a distinction between a variable and a pointer/reference to a variable. The latter allows the object to be modified by other functions by passing a reference to the original one.

In C# or Java, everything not a built-in type (int, float, sometimes String) is always a pointer or a reference. References are also garbage-collected automatically, which means they are erased when no longer used. Dynamically typed languages tend to use this memory model, too. Some Examples:

- C++:

```
void use_class(SomeClass *instance) {

    instance->use();
}

void do_something() {

    SomeClass *instance = new SomeClass; // Created as pointer.
    use_class(instance); // Passed as pointer.
    delete instance; // Otherwise it will leak memory.
}
```

- Java:

```
@Override
public final void use_class(SomeClass instance) {

    instance.use();
}

public final void do_something() {

    SomeClass instance = new SomeClass(); // Created as
reference.
    use_class(instance); // Passed as reference.
    // Garbage collector will get rid of it when not in
    // use and freeze your game randomly for a second.
}
```

- GDScript:

```
func use_class(instance): # Does not care about class type
    instance.use() # Will work with any class that has a
".use()" method.

func do_something():
    var instance = SomeClass.new() # Created as reference.
    use_class(instance) # Passed as reference.
    # Will be unreferenced and deleted.
```

In GDScript, only base types (int, float, string and the vector types) are passed by value to functions (value is copied). Everything else (instances, arrays, dictionaries, etc) is passed as reference. Classes that inherit RefCounted (the default if nothing is specified) will be freed when not used, but manual memory management is allowed too if inheriting manually from Object.

Arrays

Arrays in dynamically typed languages can contain many different mixed datatypes inside and are always dynamic (can be resized at any time). Compare for example arrays in statically typed languages:

```
int *array = new int[4]; // Create array.
array[0] = 10; // Initialize manually.
array[1] = 20; // Can't mix types.
array[2] = 40;
array[3] = 60;
// Can't resize.
use_array(array); // Passed as pointer.
delete[] array; // Must be freed.

// or

std::vector<int> array;
array.resize(4);
array[0] = 10; // Initialize manually.
array[1] = 20; // Can't mix types.
array[2] = 40;
array[3] = 60;
array.resize(3); // Can be resized.
use_array(array); // Passed reference or value.
// Freed when stack ends.
```

And in GDScript:

```
var array = [10, "hello", 40, 60] # You can mix types.
array.resize(3) # Can be resized.
```

```
use_array(array) # Passed as reference.
# Freed when no longer in use.
```

In dynamically typed languages, arrays can also double as other datatypes, such as lists:

```
var array = []
array.append(4)
array.append(5)
array.pop_front()
```

Or unordered sets:

```
var a = 20
if a in [10, 20, 30]:
    print("We have a winner!")
```

Dictionaries

Dictionaries are a powerful tool in dynamically typed languages. Most programmers that come from statically typed languages (such as C++ or C#) ignore their existence and make their life unnecessarily more difficult. This datatype is generally not present in such languages (or only in limited form).

Dictionaries can map any value to any other value with complete disregard for the datatype used as either key or value. Contrary to popular belief, they are efficient because they can be implemented with hash tables. They are, in fact, so efficient that some languages will go as far as implementing arrays as dictionaries.

Example of Dictionary:

```
var d = {"name": "John", "age": 22}
print("Name: ", d["name"], " Age: ", d["age"])
```

Dictionaries are also dynamic, keys can be added or removed at any point at little cost:

```
d["mother"] = "Rebecca" # Addition.
d["age"] = 11 # Modification.
d.erase("name") # Removal.
```

In most cases, two-dimensional arrays can often be implemented more easily with dictionaries. Here's a battleship game example:

```
# Battleship Game

const SHIP = 0
const SHIP_HIT = 1
const WATER_HIT = 2

var board = {}

func initialize():
    board[Vector2(1, 1)] = SHIP
    board[Vector2(1, 2)] = SHIP
    board[Vector2(1, 3)] = SHIP

func missile(pos):
    if pos in board: # Something at that position.
        if board[pos] == SHIP: # There was a ship! hit it.
            board[pos] = SHIP_HIT
        else:
            print("Already hit here!") # Hey dude you already hit here.
    else: # Nothing, mark as water.
        board[pos] = WATER_HIT

func game():
    initialize()
    missile(Vector2(1, 1))
    missile(Vector2(5, 8))
    missile(Vector2(2, 3))
```

Dictionaries can also be used as data markup or quick structures. While GDScript's dictionaries resemble python dictionaries, it also supports Lua style syntax and indexing, which makes it useful for writing initial states and quick structs:

```
# Same example, lua-style support.
# This syntax is a lot more readable and usable.
# Like any GDScript identifier, keys written in this form
cannot start
# with a digit.

var d = {
    name = "John",
    age = 22
}

print("Name: ", d.name, " Age: ", d.age) # Used "." based
indexing.

# Indexing

d["mother"] = "Rebecca"
d.mother = "Caroline" # This would work too to create a new
key.
```

For & while

Iterating using the C-style for loop in C-derived languages can be quite complex:

```
const char** strings = new const char*[50];

[..]

for (int i = 0; i < 50; i++) {
    printf("Value: %c Index: %d\n", strings[i], i);
}

// Even in STL:
std::list<std::string> strings;

[..]
```

```
for (std::string::const_iterator it = strings.begin(); it !=
strings.end(); it++) {
    std::cout << *it << std::endl;
}
```

Because of this, GDScript makes the opinionated decision to have a for-in loop over iterables instead:

```
for s in strings:
    print(s)
```

Container datatypes (arrays and dictionaries) are iterable. Dictionaries allow iterating the keys:

```
for key in dict:
    print(key, " -> ", dict[key])
```

Iterating with indices is also possible:

```
for i in range(strings.size()):
    print(strings[i])
```

The range() function can take 3 arguments:

```
range(n) # Will count from 0 to n in steps of 1. The parameter
n is exclusive.
range(b, n) # Will count from b to n in steps of 1. The
parameters b is inclusive. The parameter n is exclusive.
range(b, n, s) # Will count from b to n, in steps of s. The
parameters b is inclusive. The parameter n is exclusive.
```

Some examples involving C-style for loops:

```
for (int i = 0; i < 10; i++) {}

for (int i = 5; i < 10; i++) {}

for (int i = 5; i < 10; i += 2) {}
```

Translate to:

```
for i in range(10):
    pass

for i in range(5, 10):
    pass

for i in range(5, 10, 2):
    pass
```

And backwards looping done through a negative counter:

```
for (int i = 10; i > 0; i--) {}
```

Becomes:

```
for i in range(10, 0, -1):
    pass
```

While

while() loops are the same everywhere:

```
var i = 0

while i < strings.size():
    print(strings[i])
    i += 1
```

Custom iterators

You can create custom iterators in case the default ones don't quite meet your needs by overriding the Variant class's `_iter_init`, `_iter_next`, and `_iter_get` functions in your script. An example implementation of a forward iterator follows:

```
class ForwardIterator:
    var start
    var current
    var end
    var increment

    func _init(start, stop, increment):
        self.start = start
        self.current = start
        self.end = stop
        self.increment = increment

    func should_continue():
        return (current < end)

    func _iter_init(arg):
        current = start
        return should_continue()

    func _iter_next(arg):
        current += increment
        return should_continue()

    func _iter_get(arg):
        return current
```

And it can be used like any other iterator:

```
var itr = ForwardIterator.new(0, 6, 2)
for i in itr:
    print(i) # Will print 0, 2, and 4.
```

Make sure to reset the state of the iterator in `_iter_init`, otherwise nested for-loops that use custom iterators will not work as expected.

Duck typing

One of the most difficult concepts to grasp when moving from a statically typed language to a dynamic one is duck typing. Duck typing makes overall code design much simpler and straightforward to write, but it's not obvious how it works.

As an example, imagine a situation where a big rock is falling down a tunnel, smashing everything on its way. The code for the rock, in a statically typed language would be something like:

```
void BigRollingRock::on_object_hit(Smashable *entity) {

    entity->smash();
}
```

This way, everything that can be smashed by a rock would have to inherit Smashable. If a character, enemy, piece of furniture, small rock were all smashable, they would need to inherit from the class Smashable, possibly requiring multiple inheritance. If multiple inheritance was undesired, then they would have to inherit a common class like Entity. Yet, it would not be very elegant to add a virtual method `smash()` to Entity only if a few of them can be smashed.

With dynamically typed languages, this is not a problem. Duck typing makes sure you only have to define a `smash()` function where required and that's it. No need to consider inheritance, base classes, etc.

```
func _on_object_hit(object):
    object.smash()
```

And that's it. If the object that hit the big rock has a smash() method, it will be called. No need for inheritance or polymorphism. Dynamically typed languages only care about the instance having the desired method or member, not what it inherits or the class type. The definition of Duck Typing should make this clearer:

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"

In this case, it translates to:

"If the object can be smashed, don't care what it is, just smash it."

Yes, we should call it Hulk typing instead.

It's possible that the object being hit doesn't have a smash() function. Some dynamically typed languages simply ignore a method call when it doesn't exist, but GDScript is stricter, so checking if the function exists is desirable:

```
func _on_object_hit(object):
    if object.has_method("smash"):
        object.smash()
```

Then, simply define that method and anything the rock touches can be smashed.

GDScript exported properties

In Godot, class members can be exported. This means their value gets saved along with the resource (such as the scene) they're attached to. They will also be available for editing in the property editor. Exporting is done by using the `@export` annotation.

```
@export var number: int = 5
```

In that example the value `5` will be saved and visible in the property editor.

An exported variable must be initialized to a constant expression or have a type specifier in the variable. Some of the export annotations have a specific type and don't need the variable to be typed (see the *Examples* section below).

One of the fundamental benefits of exporting member variables is to have them visible and editable in the editor. This way, artists and game designers can modify values that later influence how the program runs. For this, a special export syntax is provided.

Note

Exporting properties can also be done in other languages such as C#. The syntax varies depending on the language. See C# exported properties for information on C# exports.

Basic use

If the exported value assigns a constant or constant expression, the type will be inferred and used in the editor.

```
@export var number = 5
```

If there's no default value, you can add a type to the variable.

```
@export var number: int
```

Resources and nodes can be exported.

```
@export var resource: Resource
@export var node: Node
```

Grouping Exports

It is possible to group your exported properties inside the Inspector with the @export_group annotation. Every exported property after this annotation will be added to the group. Start a new group or use `@export_group("")` to break out.

```
@export_group("My Properties")
@export var number = 3
```

The second argument of the annotation can be used to only group properties with the specified prefix.

Groups cannot be nested, use @export_subgroup to create subgroups within a group.

```
@export_subgroup("Extra Properties")
@export var string = ""
@export var flag = false
```

You can also change the name of your main category, or create additional categories in the property list with the @export_category annotation.

```
@export_category("Main Category")
@export var number = 3
@export var string = ""
```

```
@export_category("Extra Category")
@export var flag = false
```

Note

The list of properties is organized based on the class inheritance and new categories break that expectation. Use them carefully, especially when creating projects for public use.

Strings as paths

String as a path to a file.

```
@export_file var f
```

String as a path to a directory.

```
@export_dir var f
```

String as a path to a file, custom filter provided as hint.

```
@export_file("*.txt") var f
```

Using paths in the global filesystem is also possible, but only in scripts in tool mode.

String as a path to a PNG file in the global filesystem.

```
@export_global_file("*.png") var tool_image
```

String as a path to a directory in the global filesystem.

```
@export_global_dir var tool_dir
```

The multiline annotation tells the editor to show a large input field for editing over multiple lines.

```
@export_multiline var text
```

Limiting editor input ranges

Allow integer values from 0 to 20.

```
@export_range(0, 20) var i
```

Allow integer values from -10 to 20.

```
@export_range(-10, 20) var j
```

Allow floats from -10 to 20 and snap the value to multiples of 0.2.

```
@export_range(-10, 20, 0.2) var k: float
```

The limits can be only for the slider if you add the hints "or_greater" and/or "or_less".

```
@export_range(0, 100, 1, "or_greater", "or_less")
```

Floats with easing hint

Display a visual representation of the 'ease()' function when editing.

```
@export_exp_easing var transition_speed
```

Colors

Regular color given as red-green-blue-alpha value.

```
@export var col: Color
```

Color given as red-green-blue value (alpha will always be 1).

```
@export_color_no_alpha var col: Color
```

Nodes

Since Godot 4.0, nodes can be directly exported as properties in a script without having to use NodePaths:

```
# Allows any node.
@export var node: Node

# Allows any node that inherits from BaseButton.
# Custom classes declared with `class_name` can also be used.
@export var some_button: BaseButton
```

Exporting NodePaths like in Godot 3.x is still possible, in case you need it:

```
@export var node_path: NodePath
var node = get_node(node_path)
```

If you want to limit the types of nodes for NodePaths, you can use the @export_node_path annotation:

```
@export_node_path("Button", "TouchScreenButton") var
some_button
```

Resources

```
@export var resource: Resource
```

In the Inspector, you can then drag and drop a resource file from the FileSystem dock into the variable slot.

Opening the inspector dropdown may result in an extremely long list of possible classes to create, however. Therefore, if you specify an extension of Resource such as:

```
@export var resource: AnimationNode
```

The drop-down menu will be limited to AnimationNode and all its inherited classes.

It must be noted that even if the script is not being run while in the editor, the exported properties are still editable. This can be used in conjunction with a script in "tool" mode.

Exporting bit flags

Integers used as bit flags can store multiple `true`/`false` (boolean) values in one property. By using the `@export_flags` annotation, they can be set from the editor:

```
# Set any of the given flags from the editor.
@export_flags("Fire", "Water", "Earth", "Wind") var
spell_elements = 0
```

You must provide a string description for each flag. In this example, `Fire` has value 1, `Water` has value 2, `Earth` has value 4 and `Wind` corresponds to value 8. Usually, constants should be defined accordingly (e.g. `const ELEMENT_WIND = 8` and so on).

You can add explicit values using a colon:

```
@export_flags("Self:4", "Allies:8", "Foes:16") var
spell_targets = 0
```

Only power of 2 values are valid as bit flags options. The lowest allowed value is 1, as 0 means that nothing is selected. You can also add options that are a combination of other flags:

```
@export_flags("Self:4", "Allies:8", "Self and Allies:12",
"Foes:16")
```

```
var spell_targets = 0
```

Export annotations are also provided for the physics, render, and navigation layers defined in the project settings:

```
@export_flags_2d_physics var layers_2d_physics
@export_flags_2d_render var layers_2d_render
@export_flags_2d_navigation var layers_2d_navigation
@export_flags_3d_physics var layers_3d_physics
@export_flags_3d_render var layers_3d_render
@export_flags_3d_navigation var layers_3d_navigation
```

Using bit flags requires some understanding of bitwise operations. If in doubt, use boolean variables instead.

Exporting enums

Properties can be exported with a type hint referencing an enum to limit their values to the values of the enumeration. The editor will create a widget in the Inspector, enumerating the following as "Thing 1", "Thing 2", "Another Thing". The value will be stored as an integer.

```
enum NamedEnum {THING_1, THING_2, ANOTHER_THING = -1}
@export var x: NamedEnum
```

Integer and string properties can also be limited to a specific list of values using the @export_enum annotation. The editor will create a widget in the Inspector, enumerating the following as Warrior, Magician, Thief. The value will be stored as an integer, corresponding to the index of the selected option (i.e. `0`, `1`, or `2`).

```
@export_enum("Warrior", "Magician", "Thief") var
character_class: int
```

You can add explicit values using a colon:

```
@export_enum("Slow:30", "Average:60", "Very Fast:200") var
character_speed: int
```

If the type is String, the value will be stored as a string.

```
@export_enum("Rebecca", "Mary", "Leah") var character_name:
String
```

If you want to set an initial value, you must specify it explicitly:

```
@export_enum("Rebecca", "Mary", "Leah") var character_name:
String = "Rebecca"
```

Exporting arrays

Exported arrays can have initializers, but they must be constant expressions.

If the exported array specifies a type which inherits from Resource, the array values can be set in the inspector by dragging and dropping multiple files from the FileSystem dock at once.

The default value **must** be a constant expression.

```
@export var a = [1, 2, 3]
```

Exported arrays can specify type (using the same hints as before).

```
@export var ints: Array[int] = [1, 2, 3]

# Nested typed arrays such as `Array[Array[float]]` are not
supported yet.
@export var two_dimensional: Array[Array] = [[1.0, 2.0], [3.0,
4.0]]
```

You can omit the default value, but it would then be `null` if not assigned.

```
@export var b: Array
@export var scenes: Array[PackedScene]
```

Arrays with specified types which inherit from resource can be set by drag-and-dropping multiple files from the FileSystem dock.

```
@export var textures: Array[Texture] = []
@export var scenes: Array[PackedScene] = []
```

Packed type arrays also work, but only initialized empty:

```
@export var vector3s = PackedVector3Array()
@export var strings = PackedStringArray()
```

Other export variants can also be used when exporting arrays:

```
@export_range(-360, 360, 0.001, "radians") var laser_angles:
Array[float] = []
@export_file("*.json") var skill_trees: Array[String] = []
@export_color_no_alpha var hair_colors = PackedColorArray()
@export_enum("Espresso", "Mocha", "Latte", "Capuccino") var
barista_suggestions: Array[String] = []
```

`@export_storage`

By default, exporting a property has two effects:

1. makes the property stored in the scene/resource file (PROPERTY_USAGE_STORAGE);
2. adds a field to the Inspector (PROPERTY_USAGE_EDITOR).

However, sometimes you may want to make a property serializable, but not display it in the editor to prevent unintentional changes and cluttering the interface.

To do this you can use @export_storage. This can be useful for @tool scripts. Also the property value is copied when

Resource.duplicate() or Node.duplicate() is called, unlike non-exported variables.

```
var a # Not stored in the file, not displayed in the editor.
@export_storage var b # Stored in the file, not displayed in the editor.
@export var c: int # Stored in the file, displayed in the editor.
```

Setting exported variables from a tool script

When changing an exported variable's value from a script in Tool mode, the value in the inspector won't be updated automatically. To update it, call notify_property_list_changed() after setting the exported variable's value.

Advanced exports

Not every type of export can be provided on the level of the language itself to avoid unnecessary design complexity. The following describes some more or less common exporting features which can be implemented with a low-level API.

Before reading further, you should get familiar with the way properties are handled and how they can be customized with _set(), _get(), and _get_property_list() methods as described in Accessing data or logic from an object.

See also

For binding properties using the above methods in C++, see Binding properties using _set/_get/_get_property_list.

Warning

The script must operate in the `@tool` mode so the above methods can work from within the editor.

GDScript documentation comments

In GDScript, comments can be used to document your code and add descriptions to the members of a script. There are two differences between a normal comment and a documentation comment. Firstly, a documentation comment should start with double hash symbols `##`. Secondly, it must immediately precede a script member, or for script descriptions, be placed at the top of the script. If an exported variable is documented, its description is used as a tooltip in the editor. This documentation can be generated as XML files by the editor.

Documenting a script

Comments documenting a script must come before any member documentation. A suggested format for script documentation can be divided into three parts.

- A brief description of the script.
- Detailed description.
- Tutorials and deprecated/experimental marks.

To separate these from each other, the documentation comments use special tags. The tag must be at the beginning of a line (ignoring preceding white space) and must have the format `@`, followed by the keyword.

Tags

Brief description	No tag. Lives at the very beginning of the documentation section.
Description	No tag. Use one blank line to separate the description from the brief.
Tutorial	`@tutorial: https://example.com` `@tutorial(The Title Here): https://example.com`
Deprecated	`@deprecated` `@deprecated: Use [AnotherClass] instead.`

Experimental	`@experimental` `@experimental: This class is unstable.`

For example:

```
extends Node2D
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
##
## @tutorial:             https://example.com/tutorial_1
## @tutorial(Tutorial 2): https://example.com/tutorial_2
## @experimental
```

Warning

If there is any space in between the tag name and colon, for example `@tutorial  :`, it won't be treated as a valid tag and will be ignored.

Note

When the description spans multiple lines, the preceding and trailing white spaces will be stripped and joined with a single space. To preserve the line break use `[br]`. See also BBCode and class reference below.

Documenting script members

Members that are applicable for documentation:

- Inner class
- Constant
- Function
- Signal
- Variable
- Enum
- Enum value

Documentation of a script member must immediately precede the member or its annotations if it has any. The description can have more than one line but every line must start with the double hash symbol `##` to be considered as part of the documentation.

Tags

Description	No tag.
Deprecated	`@deprecated` `@deprecated: Use [member another] instead.`
Experimental	`@experimental` `@experimental: This method is incomplete.`

For example:

```
## The description of the variable.
## @deprecated: Use [member other_var] instead.
var my_var
```

Alternatively, you can use inline documentation comments:

```
enum MyEnum { ## My enum.
    VALUE_A = 0, ## Value A.
    VALUE_B = 1, ## Value B.
}

const MY_CONST = 1 ## My constant.

var my_var ## My variable.

signal my_signal ## My signal.

func my_func(): ## My func.
    pass

class MyClass: ## My class.
    pass
```

The script documentation will update in the editor help window every time the script is updated. If any member variable or function name starts with an underscore, it will be treated as private. It will not appear in the documentation and will be ignored in the help window.

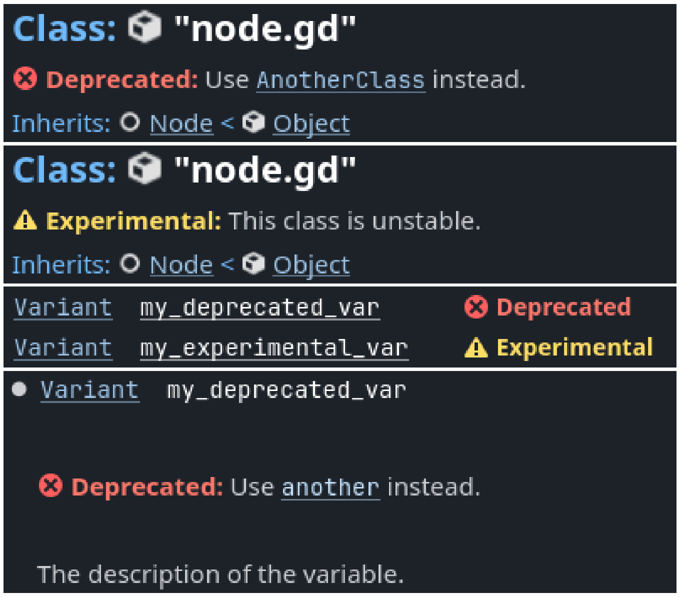
Complete script example

```
extends Node2D
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
##
## @tutorial:             https://example.com/tutorial_1
## @tutorial(Tutorial 2): https://example.com/tutorial_2
## @experimental

## The description of a constant.
const GRAVITY = 9.8

## The description of a signal.
signal my_signal

## This is a description of the below enum.
enum Direction {
    ## Direction up.
    UP = 0,
    ## Direction down.
    DOWN = 1,
    ## Direction left.
    LEFT = 2,
    ## Direction right.
    RIGHT = 3,
}

## The description of the variable v1.
var v1

## This is a multiline description of the variable v2.[br]
## The type information below will be extracted for the documentation.
var v2: int

## If the member has any annotation, the annotation should
## immediately precede it.
@export
var v3 := some_func()


## As the following function is documented, even though its name starts
with
## an underscore, it will appear in the help window.
```

```
func _fn(p1: int, p2: String) -> int:
    return 0


# The below function isn't documented and its name starts with an
underscore
# so it will treated as private and will not be shown in the help window.
func _internal() -> void:
    pass


## Documenting an inner class.
##
## The same rules apply here. The documentation must
## immediately precede the class definition.
##
## @tutorial: https://example.com/tutorial
## @experimental
class Inner:

    ## Inner class variable v4.
    var v4


    ## Inner class function fn.
    func fn(): pass
```

`@deprecated` and `@experimental` tags

You can mark a class or any of its members as deprecated or experimental. This will add the corresponding indicator in the built-in documentation viewer. Optionally, you can provide a short message explaining why the API is not recommended. This can be especially useful for plugin and library creators.



Class: "node.gd"

Deprecated: Use AnotherClass instead.

Inherits: Node < Object

Class: "node.gd"

Experimental: This class is unstable.

Inherits: Node < Object

`Variant`	`my_deprecated_var`	**Deprecated**
`Variant`	`my_experimental_var`	**Experimental**

● `Variant` `my_deprecated_var`

Deprecated: Use `another` instead.

The description of the variable.

- **Deprecated** marks a non-recommended API that is subject to removal or incompatible change in a future major release. Usually the API is kept for backwards compatibility.
- **Experimental** marks a new unstable API that may be changed or removed in the current major branch. Using this API is not recommended in production code.

Note

While technically you can use both `@deprecated` and `@experimental` tags on the same class/member, this is not recommended as it is against common conventions.

BBCode and class reference

Godot's class reference supports BBCode-like tags. They add nice formatting to the text which could also be used in the documentation. See also class reference bbcode. Note that this is slightly different from the `RichTextLabel` BBCode.

Whenever you link to a member of another class, you need to specify the class name. For links to the same class, the class name is optional and can be omitted.

Here's the list of available tags:

Tag and Description	Example	Result
`[Class]` Link to class	`Move the [Sprite2D].`	Move the Sprite2D.
`[annotation Class.name]` Link to annotation	`See [annotation @GDScript.@rpc].`	See @GDScript.@rpc.
`[constant Class.name]` Link to constant	`See [constant Color.RED].`	See Color.RED.
`[enum Class.name]` Link to enum	`See [enum Mesh.ArrayType].`	See Mesh.ArrayType.
`[member Class.name]` Link to member (property)	`Get [member Node2D.scale].`	Get Node2D.scale.
`[method Class.name]` Link to method	`Call [method Node3D.hide].`	Call Node3D.hide().
`[constructor Class.name]` Link to built-in constructor	`Use [constructor Color.Color].`	Use Color.Color.
`[operator Class.name]`	`Use [operator Color.operator *].`	Use Color.operator *.

Tag and Description	**Example**	**Result**
Link to built-in operator		
`[signal Class.name]` Link to signal	`Emit [signal Node.renamed].`	Emit Node.renamed.
`[theme_item Class.name]` Link to theme item	`See [theme_item Label.font].`	See Label.font.
`[param name]` Parameter name (as code)	`Takes [param size] for the size.`	Takes `size` for the size.
`[br]` Line break	`Line 1.[br]` `Line 2.`	Line 1. Line 2.
`[lb]` `[rb]` `[` and `]` respectively	`[lb]b[rb]text[lb]/b[rb]`	[b]text[/b]
`[b]` `[/b]` Bold	`Do [b]not[/b] call this method.`	Do **not** call this method.
`[i]` `[/i]` Italic	`Returns the [i]global[/i] position.`	Returns the *global* position.
`[u]` `[/u]` Underline	`[u]Always[/u] use this method.`	Always use this method.

Tag and Description	**Example**	**Result**
`[s] [/s]` Strikethrough	`[s]Outdated information.[/s]`	~~Outdated information.~~

`[color]`
`[/color]`
Color
	`[color=red]Error![/color]`	Error!

`[font] [/font]` Font	`[font=res://mono.ttf]LICENSE[/font]`	`LICENSE`
`[img] [/img]` Image	`[img width=32]res://icon.svg[/img]`	
`[url] [/url]` Hyperlink	`[url]https://example.com[/url]` `[url=https://example.com]Website[/url]`	https://example.com Website

`[center]`
`[/center]`
Horizontal centering
	`[center]2 + 2 = 4[/center]`	2 + 2 = 4

`[kbd] [/kbd]` Keyboard/mouse shortcut	`Press [kbd]Ctrl + C[/kbd].`	Press `Ctrl + C`.
`[code] [/code]` Inline code fragment	`Returns [code]true[/code].`	Returns `true`.
`[codeblock]` `[/codeblock]`	*See below.*	*See below.*

Tag and Description	**Example**	**Result**
Multiline code block		

Note

1. Currently only @GDScript has annotations.
2. `[kbd]` disables BBCode until the parser encounters `[/kbd]`.
3. `[code]` disables BBCode until the parser encounters `[/code]`.
4. `[codeblock]` disables BBCode until the parser encounters `[/codeblock]`.

Warning

Use `[codeblock]` for pre-formatted code blocks. Inside `[codeblock]`, always use **four spaces** for indentation (the parser will delete tabs).

```
## Do something for this plugin. Before using the method
## you first have to [method initialize] [MyPlugin].[br]
## [color=yellow]Warning:[/color] Always [method clean] after use.[br]
## Usage:
## [codeblock]
## func _ready():
##     the_plugin.initialize()
##     the_plugin.do_something()
##     the_plugin.clean()
## [/codeblock]
func do_something():
    pass
```

By default, `[codeblock]` highlights GDScript syntax. You can change it using the `lang` attribute. Currently supported options are:

- `[codeblock lang=text]` disables syntax highlighting;
- `[codeblock lang=gdscript]` highlights GDScript syntax;
- `[codeblock lang=csharp]` highlights C# syntax (only in .NET version).

GDScript style guide

This style guide lists conventions to write elegant GDScript. The goal is to encourage writing clean, readable code and promote consistency across projects, discussions, and tutorials. Hopefully, this will also support the development of auto-formatting tools.

Since GDScript is close to Python, this guide is inspired by Python's PEP 8 [https://www.python.org/dev/peps/pep-0008/] programming style guide.

Style guides aren't meant as hard rulebooks. At times, you may not be able to apply some of the guidelines below. When that happens, use your best judgment, and ask fellow developers for insights.

In general, keeping your code consistent in your projects and within your team is more important than following this guide to a tee.

Note

Godot's built-in script editor uses a lot of these conventions by default. Let it help you.

Here is a complete class example based on these guidelines:

```
class_name StateMachine
extends Node
## Hierarchical State machine for the player.
##
## Initializes states and delegates engine callbacks ([method
Node._physics_process],
## [method Node._unhandled_input]) to the state.

signal state_changed(previous, new)
```

```
@export var initial_state: Node
var is_active = true:
    set = set_is_active

@onready var _state = initial_state:
    set = set_state
@onready var _state_name = _state.name


func _init():
    add_to_group("state_machine")


func _enter_tree():
    print("this happens before the ready method!")


func _ready():
    state_changed.connect(_on_state_changed)
    _state.enter()


func _unhandled_input(event):
    _state.unhandled_input(event)


func _physics_process(delta):
    _state.physics_process(delta)


func transition_to(target_state_path, msg={}):
    if not has_node(target_state_path):
        return

    var target_state = get_node(target_state_path)
    assert(target_state.is_composite == false)

    _state.exit()
    self._state = target_state
    _state.enter(msg)
    Events.player_state_changed.emit(_state.name)
```

```
func set_is_active(value):
    is_active = value
    set_physics_process(value)
    set_process_unhandled_input(value)
    set_block_signals(not value)


func set_state(value):
    _state = value
    _state_name = _state.name


func _on_state_changed(previous, new):
    print("state changed")
    state_changed.emit()


class State:
    var foo = 0

    func _init():
        print("Hello!")
```

Formatting

Encoding and special characters

- Use line feed (**LF**) characters to break lines, not CRLF or CR. *(editor default)*
- Use one line feed character at the end of each file. *(editor default)*
- Use **UTF-8** encoding without a byte order mark [https://en.wikipedia.org/wiki/Byte_order_mark]. *(editor default)*
- Use **Tabs** instead of spaces for indentation. *(editor default)*

Indentation

Each indent level should be one greater than the block containing it.

Good:

```
for i in range(10):
    print("hello")
```

Bad:

```
for i in range(10):
  print("hello")

for i in range(10):
        print("hello")
```

Use 2 indent levels to distinguish continuation lines from regular code blocks.

Good:

```
effect.interpolate_property(sprite, "transform/scale",
        sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
        Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Bad:

```
effect.interpolate_property(sprite, "transform/scale",
    sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
    Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Exceptions to this rule are arrays, dictionaries, and enums. Use a single indentation level to distinguish continuation lines:

Good:

```
var party = [
    "Godot",
    "Godette",
    "Steve",
]
```

```
var character_dict = {
    "Name": "Bob",
    "Age": 27,
    "Job": "Mechanic",
}

enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT,
}
```

Bad:

```
var party = [
        "Godot",
        "Godette",
        "Steve",
]

var character_dict = {
        "Name": "Bob",
        "Age": 27,
        "Job": "Mechanic",
}

enum Tiles {
        TILE_BRICK,
        TILE_FLOOR,
        TILE_SPIKE,
        TILE_TELEPORT,
}
```

Trailing comma

Use a trailing comma on the last line in arrays, dictionaries, and enums. This results in easier refactoring and better diffs in version control as the last line doesn't need to be modified when adding new elements.

Good:

```
var array = [
    1,
    2,
    3,
]
```

Bad:

```
var array = [
    1,
    2,
    3
]
```

Trailing commas are unnecessary in single-line lists, so don't add them in this case.

Good:

```
var array = [1, 2, 3]
```

Bad:

```
var array = [1, 2, 3,]
```

Blank lines

Surround functions and class definitions with two blank lines:

```
func heal(amount):
    health += amount
    health = min(health, max_health)
    health_changed.emit(health)


func take_damage(amount, effect=null):
    health -= amount
    health = max(0, health)
    health_changed.emit(health)

```

Use one blank line inside functions to separate logical sections.

Note

We use a single line between classes and function definitions in the class reference and in short code snippets in this documentation.

Line length

Keep individual lines of code under 100 characters.

If you can, try to keep lines under 80 characters. This helps to read the code on small displays and with two scripts opened side-by-side in an external text editor. For example, when looking at a differential revision.

One statement per line

Avoid combining multiple statements on a single line, including conditional statements, to adhere to the GDScript style guidelines for readability.

Good:

```
if position.x > width:
    position.x = 0

if flag:
    print("flagged")
```

Bad:

```
if position.x > width: position.x = 0

if flag: print("flagged")
```

The only exception to that rule is the ternary operator:

```
next_state = "idle" if is_on_floor() else "fall"
```

Format multiline statements for readability

When you have particularly long `if` statements or nested ternary expressions, wrapping them over multiple lines improves readability. Since continuation lines are still part of the same expression, 2 indent levels should be used instead of one.

GDScript allows wrapping statements using multiple lines using parentheses or backslashes. Parentheses are favored in this style guide since they make for easier refactoring. With backslashes, you have to ensure that the last line never contains a backslash at the end. With parentheses, you don't have to worry about the last line having a backslash at the end.

When wrapping a conditional expression over multiple lines, the `and`/`or` keywords should be placed at the beginning of the line continuation, not at the end of the previous line.

Good:

```
var angle_degrees = 135
var quadrant = (
        "northeast" if angle_degrees <= 90
        else "southeast" if angle_degrees <= 180
        else "southwest" if angle_degrees <= 270
        else "northwest"
)

var position = Vector2(250, 350)
if (
        position.x > 200 and position.x < 400
        and position.y > 300 and position.y < 400
):
    pass
```

Bad:

```
var angle_degrees = 135
var quadrant = "northeast" if angle_degrees <= 90 else
"southeast" if angle_degrees <= 180 else "southwest" if
angle_degrees <= 270 else "northwest"

var position = Vector2(250, 350)
if position.x > 200 and position.x < 400 and position.y > 300
and position.y < 400:
    pass
```

Avoid unnecessary parentheses

Avoid parentheses in expressions and conditional statements. Unless necessary for order of operations or wrapping over multiple lines, they only reduce readability.

Good:

```
if is_colliding():
    queue_free()
```

Bad:

```
if (is_colliding()):
    queue_free()
```

Boolean operators

Prefer the plain English versions of boolean operators, as they are the most accessible:

- Use `and` instead of `&&`.
- Use `or` instead of `||`.
- Use `not` instead of `!`.

You may also use parentheses around boolean operators to clear any ambiguity. This can make long expressions easier to read.

Good:

```
if (foo and bar) or not baz:
    print("condition is true")
```

Bad:

```
if foo && bar || !baz:
    print("condition is true")
```

Comment spacing

Regular comments (`#`) and documentation comments (`##`) should start with a space, but not code that you comment out. Additionally, code region comments (`#region`/`#endregion`) must follow that precise syntax, so they should not start with a space.

Using a space for regular and documentation comments helps differentiate text comments from disabled code.

Good:

```
# This is a comment.
#print("This is disabled code")
```

Bad:

```
#This is a comment.
# print("This is disabled code")
```

Note

In the script editor, to toggle commenting of the selected code, press `Ctrl + K`. This feature adds/removes a single `#` sign before any code on the selected lines.

Whitespace

Always use one space around operators and after commas. Also, avoid extra spaces in dictionary references and function calls. One exception to this is for single-line dictionary declarations, where a space should be added after the opening brace and before the closing brace. This makes the dictionary easier to visually distinguish from an array, as the `[]` characters look close to `{}` with most fonts.

Good:

```
position.x = 5
position.y = target_position.y + 10
dict["key"] = 5
my_array = [4, 5, 6]
my_dictionary = { key = "value" }
print("foo")
```

Bad:

```
position.x=5
position.y = mpos.y+10
dict ["key"] = 5
myarray = [4,5,6]
my_dictionary = {key = "value"}
print ("foo")
```

Don't use spaces to align expressions vertically:

```
x        = 100
y        = 100
velocity = 500
```

Quotes

Use double quotes unless single quotes make it possible to escape fewer characters in a given string. See the examples below:

```
# Normal string.
print("hello world")

# Use double quotes as usual to avoid escapes.
print("hello 'world'")

# Use single quotes as an exception to the rule to avoid
escapes.
print('hello "world"')

# Both quote styles would require 2 escapes; prefer double
quotes if it's a tie.
print("'hello' \"world\"")
```

Numbers

Don't omit the leading or trailing zero in floating-point numbers. Otherwise, this makes them less readable and harder to distinguish from integers at a glance.

Good:

```
var float_number = 0.234
var other_float_number = 13.0
```

Bad:

```
var float_number = .234
var other_float_number = 13.
```

Use lowercase for letters in hexadecimal numbers, as their lower height makes the number easier to read.

Good:

```
var hex_number = 0xfb8c0b
```

Bad:

```
var hex_number = 0xFB8C0B
```

Take advantage of GDScript's underscores in literals to make large numbers more readable.

Good:

```
var large_number = 1_234_567_890
var large_hex_number = 0xffff_f8f8_0000
var large_bin_number = 0b1101_0010_1010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12345
```

Bad:

```
var large_number = 1234567890
var large_hex_number = 0xffffff8f80000
var large_bin_number = 0b110100101010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12_345
```

Naming conventions

These naming conventions follow the Godot Engine style. Breaking these will make your code clash with the built-in naming conventions, leading to inconsistent code.

File names

Use snake_case for file names. For named classes, convert the PascalCase class name to snake_case:

```
# This file should be saved as `weapon.gd`.
class_name Weapon
```

```
extends Node

```

```
# This file should be saved as `yaml_parser.gd`.
class_name YAMLParser
extends Object
```

This is consistent with how C++ files are named in Godot's source code. This also avoids case sensitivity issues that can crop up when exporting a project from Windows to other platforms.

Classes and nodes

Use PascalCase for class and node names:

```
extends CharacterBody3D
```

Also use PascalCase when loading a class into a constant or a variable:

```
const Weapon = preload("res://weapon.gd")
```

Functions and variables

Use snake_case to name functions and variables:

```
var particle_effect
func load_level():
```

Prepend a single underscore (_) to virtual methods functions the user must override, private functions, and private variables:

```
var _counter = 0
func _recalculate_path():
```

Signals

Use the past tense to name signals:

```
signal door_opened
signal score_changed
```

Constants and enums

Write constants with CONSTANT_CASE, that is to say in all caps with an underscore (_) to separate words:

```
const MAX_SPEED = 200
```

Use PascalCase for enum *names* and CONSTANT_CASE for their members, as they are constants:

```
enum Element {
    EARTH,
    WATER,
    AIR,
    FIRE,
}
```

Write enums with each item on its own line. This allows adding documentation comments above each item more easily, and also makes for cleaner diffs in version control when items are added or removed.

Good:

```
enum Element {
    EARTH,
    WATER,
    AIR,
    FIRE,
}
```

Bad:

```
enum Element { EARTH, WATER, AIR, FIRE }
```

Code order

This section focuses on code order. For formatting, see Formatting. For naming conventions, see Naming conventions.

We suggest to organize GDScript code this way:

```
01. @tool
02. class_name
03. extends
04. # docstring

05. signals
06. enums
07. constants
08. @export variables
09. public variables
10. private variables
11. @onready variables

12. optional built-in virtual _init method
13. optional built-in virtual _enter_tree() method
14. built-in virtual _ready method
15. remaining built-in virtual methods
16. public methods
17. private methods
18. subclasses
```

We optimized the order to make it easy to read the code from top to bottom, to help developers reading the code for the first time understand how it works, and to avoid errors linked to the order of variable declarations.

This code order follows four rules of thumb:

1. Properties and signals come first, followed by methods.
2. Public comes before private.
3. Virtual callbacks come before the class's interface.

4. The object's construction and initialization functions, `_init` and `_ready`, come before functions that modify the object at runtime.

Class declaration

If the code is meant to run in the editor, place the `@tool` annotation on the first line of the script.

Follow with the `class_name` if necessary. You can turn a GDScript file into a global type in your project using this feature. For more information, see GDScript reference.

Then, add the `extends` keyword if the class extends a built-in type.

Following that, you should have the class's optional documentation comments. You can use that to explain the role of your class to your teammates, how it works, and how other developers should use it, for example.

```
class_name MyNode
extends Node
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
```

Signals and properties

Write signal declarations, followed by properties, that is to say, member variables, after the docstring.

Enums should come after signals, as you can use them as export hints for other properties.

Then, write constants, exported variables, public, private, and onready variables, in that order.

```
signal player_spawned(position)

enum Jobs {
    KNIGHT,
    WIZARD,
    ROGUE,
    HEALER,
    SHAMAN,
}

const MAX_LIVES = 3

@export var job: Jobs = Jobs.KNIGHT
@export var max_health = 50
@export var attack = 5

var health = max_health:
    set(new_health):
        health = new_health

var _speed = 300.0

@onready var sword = get_node("Sword")
@onready var gun = get_node("Gun")
```

Note

GDScript evaluates `@onready` variables right before the `_ready` callback. You can use that to cache node dependencies, that is to say, to get child nodes in the scene that your class relies on. This is what the example above shows.

Member variables

Don't declare member variables if they are only used locally in a method, as it makes the code more difficult to follow. Instead, declare them as local variables in the method's body.

Local variables

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Methods and static functions

After the class's properties come the methods.

Start with the `_init()` callback method, that the engine will call upon creating the object in memory. Follow with the `_ready()` callback, that Godot calls when it adds a node to the scene tree.

These functions should come first because they show how the object is initialized.

Other built-in virtual callbacks, like `_unhandled_input()` and `_physics_process`, should come next. These control the object's main loop and interactions with the game engine.

The rest of the class's interface, public and private methods, come after that, in that order.

```
func _init():
    add_to_group("state_machine")


func _ready():
    state_changed.connect(_on_state_changed)
    _state.enter()
```

```
func _unhandled_input(event):
    _state.unhandled_input(event)


func transition_to(target_state_path, msg={}):
    if not has_node(target_state_path):
        return

    var target_state = get_node(target_state_path)
    assert(target_state.is_composite == false)

    _state.exit()
    self._state = target_state
    _state.enter(msg)
    Events.player_state_changed.emit(_state.name)


func _on_state_changed(previous, new):
    print("state changed")
    state_changed.emit()
```

Static typing

Since Godot 3.1, GDScript supports optional static typing.

Declared types

To declare a variable's type, use `<variable>: <type>`:

```
var health: int = 0
```

To declare the return type of a function, use `-> <type>`:

```
func heal(amount: int) -> void:
```

Inferred types

In most cases you can let the compiler infer the type, using `:=`. Prefer `:=` when the type is written on the same line as the assignment, otherwise prefer writing the type explicitly.

Good:

```
var health: int = 0 # The type can be int or float, and thus
should be stated explicitly.
var direction := Vector3(1, 2, 3) # The type is clearly
inferred as Vector3.
```

Include the type hint when the type is ambiguous, and omit the type hint when it's redundant.

Bad:

```
var health := 0 # Typed as int, but it could be that float was
intended.
var direction: Vector3 = Vector3(1, 2, 3) # The type hint has
redundant information.

# What type is this? It's not immediately clear to the reader,
so it's bad.
var value := complex_function()
```

In some cases, the type must be stated explicitly, otherwise the behavior will not be as expected because the compiler will only be able to use the function's return type. For example, `get_node()` cannot infer a type unless the scene or file of the node is loaded in memory. In this case, you should set the type explicitly.

Good:

```
@onready var health_bar: ProgressBar = get_node("UI/LifeBar")
```

Alternatively, you can use the `as` keyword to cast the return type, and that type will be used to infer the type of the var.

```
@onready var health_bar := get_node("UI/LifeBar") as
ProgressBar
```

```
# health_bar will be typed as ProgressBar
```

This option is also considered more type-safe than the first.

Bad:

```
# The compiler can't infer the exact type and will use Node
# instead of ProgressBar.
@onready var health_bar := get_node("UI/LifeBar")
```

Static typing in GDScript

In this guide, you will learn:

- how to use static typing in GDScript;
- that static types can help you avoid bugs;
- that static typing improves your experience with the editor.

Where and how you use this language feature is entirely up to you: you can use it only in some sensitive GDScript files, use it everywhere, or don't use it at all.

Static types can be used on variables, constants, functions, parameters, and return types.

A brief look at static typing

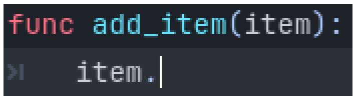
With static typing, GDScript can detect more errors without even running the code. Also type hints give you and your teammates more information as you're working, as the arguments' types show up when you call a method. Static typing improves editor autocompletion and documentation of your scripts.

Imagine you're programming an inventory system. You code an `Item` class, then an `Inventory`. To add items to the inventory, the people who work with your code should always pass an `Item` to the `Inventory.add()` method. With types, you can enforce this:

```
class_name Inventory


func add(reference: Item, amount: int = 1):
    var item := find_item(reference)
    if not item:
        item = _instance_item_from_db(reference)
    item.amount += amount
```

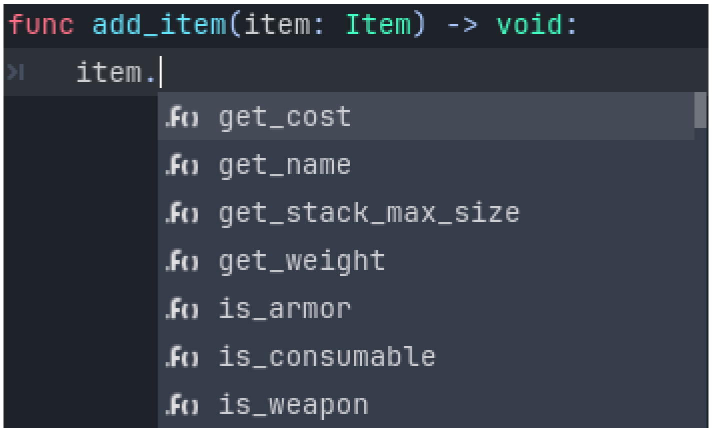
Static types also give you better code completion options. Below, you can see the difference between a dynamic and a static typed completion options.

You've probably encountered a lack of autocomplete suggestions after a dot:



```
func add_item(item):
	item.
```

This is due to dynamic code. Godot cannot know what value type you're passing to the function. If you write the type explicitly however, you will get all methods, properties, constants, etc. from the value:



```
func add_item(item: Item) -> void:
	item.
```

Tip

If you prefer static typing, we recommend enabling the **Text Editor > Completion > Add Type Hints** editor setting. Also consider enabling some warnings that are disabled by default.

Also, typed GDScript improves performance by using optimized opcodes when operand/argument types are known at compile time. More GDScript optimizations are planned in the future, such as JIT/AOT compilation.

Overall, typed programming gives you a more structured experience. It helps prevent errors and improves the self-documenting aspect of your scripts. This is especially helpful when you're working in a team or on a long-term project: studies have shown that developers spend most of their time reading other people's code, or scripts they wrote in the past and forgot about. The clearer and the more structured the code, the faster it is to understand, the faster you can move forward.

How to use static typing

To define the type of a variable, parameter, or constant, write a colon after the name, followed by its type. E.g. `var health: int`. This forces the variable's type to always stay the same:

```
var damage: float = 10.5
const MOVE_SPEED: float = 50.0
func sum(a: float = 0.0, b: float = 0.0) -> float:
    return a + b
```

Godot will try to infer types if you write a colon, but you omit the type:

```
var damage := 10.5
const MOVE_SPEED := 50.0
func sum(a := 0.0, b := 0.0) -> float:
    return a + b
```

Note

1. There is no difference between `=` and `:=` for constants.

2. You don't need to write type hints for constants, as Godot sets it automatically from the assigned value. But you can still do so to make the intent of your code clearer. Also, this is useful for typed arrays (like `const A: Array[int] = [1, 2, 3]`), since untyped arrays are used by default.

What can be a type hint

Here is a complete list of what can be used as a type hint:

1. `Variant`. Any type. In most cases this is not much different from an untyped declaration, but increases readability. As a return type, forces the function to explicitly return some value.
2. *(Only return type)* `void`. Indicates that the function does not return any value.
3. Built-in types.
4. Native classes (`Object`, `Node`, `Area2D`, `Camera2D`, etc.).
5. Global classes.
6. Inner classes.
7. Global, native and custom named enums. Note that an enum type is just an `int`, there is no guarantee that the value belongs to the set of enum values.
8. Constants (including local ones) if they contain a preloaded class or enum.

You can use any class, including your custom classes, as types. There are two ways to use them in scripts. The first method is to preload the script you want to use as a type in a constant:

```
const Rifle = preload("res://player/weapons/rifle.gd")
var my_rifle: Rifle
```

The second method is to use the `class_name` keyword when you create. For the example above, your `rifle.gd` would look like this:

```
class_name Rifle
extends Node2D
```

If you use `class_name`, Godot registers the `Rifle` type globally in the editor, and you can use it anywhere, without having to preload it into a constant:

```
var my_rifle: Rifle
```

Specify the return type of a function with the arrow `->`

To define the return type of a function, write a dash and a right angle bracket `->` after its declaration, followed by the return type:

```
func _process(delta: float) -> void:
    pass
```

The type `void` means the function does not return anything. You can use any type, as with variables:

```
func hit(damage: float) -> bool:
    health_points -= damage
    return health_points <= 0
```

You can also use your own classes as return types:

```
# Adds an item to the inventory and returns it.
func add(reference: Item, amount: int) -> Item:
    var item: Item = find_item(reference)
    if not item:
        item = ItemDatabase.get_instance(reference)

    item.amount += amount
    return item
```

Covariance and contravariance

When inheriting base class methods, you should follow the Liskov substitution principle [https://en.wikipedia.org/wiki/Liskov_substitution_principle].

Covariance: When you inherit a method, you can specify a return type that is more specific (**subtype**) than the parent method.

Contravariance: When you inherit a method, you can specify a parameter type that is less specific (**supertype**) than the parent method.

Example:

```
class_name Parent


func get_property(param: Label) -> Node:
    # ...
```

```
class_name Child extends Parent


# `Control` is a supertype of `Label`.
# `Node2D` is a subtype of `Node`.
func get_property(param: Control) -> Node2D:
    # ...
```

Specify the element type of an `Array`

To define the type of an `Array`, enclose the type name in `[]`.

An array's type applies to `for` loop variables, as well as some operators like `[]`, `[]=`, and `+`. Array methods (such as `push_back`) and other operators (such as `==`) are still untyped. Built-in types, native and custom classes, and enums may be used as element types. Nested array types are not supported.

```
var scores: Array[int] = [10, 20, 30]
var vehicles: Array[Node] = [$Car, $Plane]
var items: Array[Item] = [Item.new()]
# var arrays: Array[Array] -- disallowed

for score in scores:
    # score has type `int`

# The following would be errors:
scores += vehicles
var s: String = scores[0]
scores[0] = "lots"
```

Since Godot 4.2, you can also specify a type for the loop variable in a `for` loop. For instance, you can write:

```
var names = ["John", "Marta", "Samantha", "Jimmy"]
for name: String in names:
    pass
```

The array will remain untyped, but the `name` variable within the `for` loop will always be of `String` type.

Type casting

Type casting is an important concept in typed languages. Casting is the conversion of a value from one type to another.

Imagine an `Enemy` in your game, that `extends Area2D`. You want it to collide with the `Player`, a `CharacterBody2D` with a script called `PlayerController` attached to it. You use the `body_entered` signal to detect the collision. With typed code, the body you detect is going to be a generic `PhysicsBody2D`, and not your `PlayerController` on the `_on_body_entered` callback.

You can check if this `PhysicsBody2D` is your `Player` with the `as` keyword, and using the colon `:` again to force the variable to use this type. This forces the variable to stick to the `PlayerController` type:

```
func _on_body_entered(body: PhysicsBody2D) -> void:
    var player := body as PlayerController
    if not player:
        return

    player.damage()
```

As we're dealing with a custom type, if the `body` doesn't extend `PlayerController`, the `player` variable will be set to `null`. We can use this to check if the body is the player or not. We will also get full autocompletion on the player variable thanks to that cast.

Note

The `as` keyword silently casts the variable to `null` in case of a type mismatch at runtime, without an error/warning. While this may be convenient in some cases, it can also lead to bugs. Use the `as` keyword only if this behavior is intended. A safer alternative is to use the `is` keyword:

```
if not (body is PlayerController):
    push_error("Bug: body is not PlayerController.")

var player: PlayerController = body
if not player:
    return

player.damage()
```

or `assert()` statement:

```
assert(body is PlayerController, "Bug: body is not
PlayerController.")

var player: PlayerController = body
if not player:
    return

player.damage()
```

Note

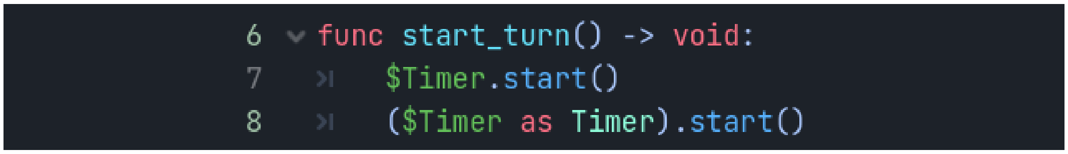
If you try to cast with a built-in type and it fails, Godot will throw an error.

Safe lines

You can also use casting to ensure safe lines. Safe lines are a tool to tell you when ambiguous lines of code are type-safe. As you can mix and match typed and dynamic code, at times, Godot doesn't have enough information to know if an instruction will trigger an error or not at runtime.

This happens when you get a child node. Let's take a timer for example: with dynamic code, you can get the node with `$Timer`. GDScript supports duck-typing [https://stackoverflow.com/a/4205163/8125343], so even if your timer is of type `Timer`, it is also a `Node` and an `Object`, two classes it extends. With dynamic GDScript, you also don't care about the node's type as long as it has the methods you need to call.

You can use casting to tell Godot the type you expect when you get a node: `($Timer as Timer)`, `($Player as CharacterBody2D)`, etc. Godot will ensure the type works and if so, the line number will turn green at the left of the script editor.



```
func start_turn() -> void:
	$Timer.start()
	($Timer as Timer).start()
```

Unsafe line (line 7) vs Safe Lines (line 6 and 8)

Note

Safe lines do not always mean better or more reliable code. See the note above about the `as` keyword. For example:

```
@onready var node_1 := $Node1 as Type1 # Safe line.
@onready var node_2: Type2 = $Node2 # Unsafe line.
```

Even though `node_2` declaration is marked as an unsafe line, it is more reliable than `node_1` declaration. Because if you change the node type in the scene and accidentally forget to change it in the script, the error will be detected immediately when the scene is loaded. Unlike `node_1`, which will be silently cast to `null` and the error will be detected later.

Note

You can turn off safe lines or change their color in the editor settings.

Typed or dynamic: stick to one style

Typed GDScript and dynamic GDScript can coexist in the same project. But it's recommended to stick to either style for consistency in your codebase, and for your peers. It's easier for everyone to work together if you follow the same guidelines, and faster to read and understand other people's code.

Typed code takes a little more writing, but you get the benefits we discussed above. Here's an example of the same, empty script, in a dynamic style:

```
extends Node


func _ready():
    pass


func _process(delta):
    pass
```

And with static typing:

```
extends Node


func _ready() -> void:
    pass


func _process(delta: float) -> void:
    pass
```

As you can see, you can also use types with the engine's virtual methods. Signal callbacks, like any methods, can also use types. Here's a `body_entered` signal in a dynamic style:

```
func _on_area_2d_body_entered(body):
    pass
```

And the same callback, with type hints:

```
func _on_area_entered(area: CollisionObject2D) -> void:
    pass
```

Warning system

Note

Detailed documentation about the GDScript warning system has been moved to GDScript warning system.

From version 3.1, Godot gives you warnings about your code as you write it: the engine identifies sections of your code that may lead to issues at runtime, but lets you decide whether or not you want to leave the code as it is.

We have a number of warnings aimed specifically at users of typed GDScript. By default, these warnings are disabled, you can enable them in Project Settings (**Debug > GDScript**, make sure **Advanced Settings** is enabled).

You can enable the `UNTYPED_DECLARATION` warning if you want to always use static types. Additionally, you can enable the `INFERRED_DECLARATION` warning if you prefer a more readable and reliable, but more verbose syntax.

`UNSAFE_*` warnings make unsafe operations more noticeable, than unsafe lines. Currently, `UNSAFE_*` warnings do not cover all cases that unsafe lines cover.

Common unsafe operations and their safe counterparts

`UNSAFE_PROPERTY_ACCESS` and `UNSAFE_METHOD_ACCESS` warnings

In this example, we aim to set a property and call a method on an object that has a script attached with `class_name MyScript` and that `extends Node2D`. If we have a reference to the object as a `Node2D` (for instance, as it was passed to us by the physics system), we can first check if the property and method exist and then set and call them if they do:

```
if "some_property" in node_2d:
    node_2d.some_property = 20  # Produces UNSAFE_PROPERTY_ACCESS warning.

if node_2d.has_method("some_function"):
    node_2d.some_function()  # Produces UNSAFE_METHOD_ACCESS warning.
```

However, this code will produce `UNSAFE_PROPERTY_ACCESS` and `UNSAFE_METHOD_ACCESS` warnings as the property and method are not present in the referenced type - in this case a `Node2D`. To make these operations safe, you can first check if the object is of type `MyScript` using the `is` keyword and then declare a variable with the type `MyScript` on which you can set its properties and call its methods:

```
if node_2d is MyScript:
    var my_script: MyScript = node_2d
    my_script.some_property = 20
    my_script.some_function()
```

Alternatively, you can declare a variable and use the `as` operator to try to cast the object. You'll then want to check whether the cast was successful by confirming that the variable was assigned:

```
var my_script := node_2d as MyScript
if my_script != null:
    my_script.some_property = 20
    my_script.some_function()
```

`UNSAFE_CAST` warning

In this example, we would like the label connected to an object entering our collision area to show the area's name. Once the object enters the collision area, the physics system sends a signal with a

`Node2D` object, and the most straightforward (but not statically typed) solution to do what we want could be achieved like this:

```
func _on_body_entered(body: Node2D) -> void:
    body.label.text = name  # Produces UNSAFE_PROPERTY_ACCESS
warning.
```

This piece of code produces an `UNSAFE_PROPERTY_ACCESS` warning because `label` is not defined in `Node2D`. To solve this, we could first check if the `label` property exist and cast it to type `Label` before settings its text property like so:

```
func _on_body_entered(body: Node2D) -> void:
    if "label" in body:
        (body.label as Label).text = name  # Produces
UNSAFE_CAST warning.
```

However, this produces an `UNSAFE_CAST` warning because `body.label` is of a `Variant` type. To safely get the property in the type you want, you can use the `Object.get()` method which returns the object as a `Variant` value or returns `null` if the property doesn't exist. You can then determine whether the property contains an object of the right type using the `is` keyword, and finally declare a statically typed variable with the object:

```
func _on_body_entered(body: Node2D) -> void:
    var label_variant: Variant = body.get("label")
    if label_variant is Label:
        var label: Label = label_variant
        label.text = name
```

Cases where you can't specify types

To wrap up this introduction, let's mention cases where you can't use type hints. This will trigger a **syntax error**.

1. You can't specify the type of individual elements in an array or a dictionary:

```
var enemies: Array = [$Goblin: Enemy, $Zombie: Enemy]
var character: Dictionary = {
    name: String = "Richard",
    money: int = 1000,
    inventory: Inventory = $Inventory,
}
```

2. Nested types are not currently supported:

```
var teams: Array[Array[Character]] = []
```

3. Typed dictionaries are not currently supported:

```
var map: Dictionary[Vector2i, Item] = {}
```

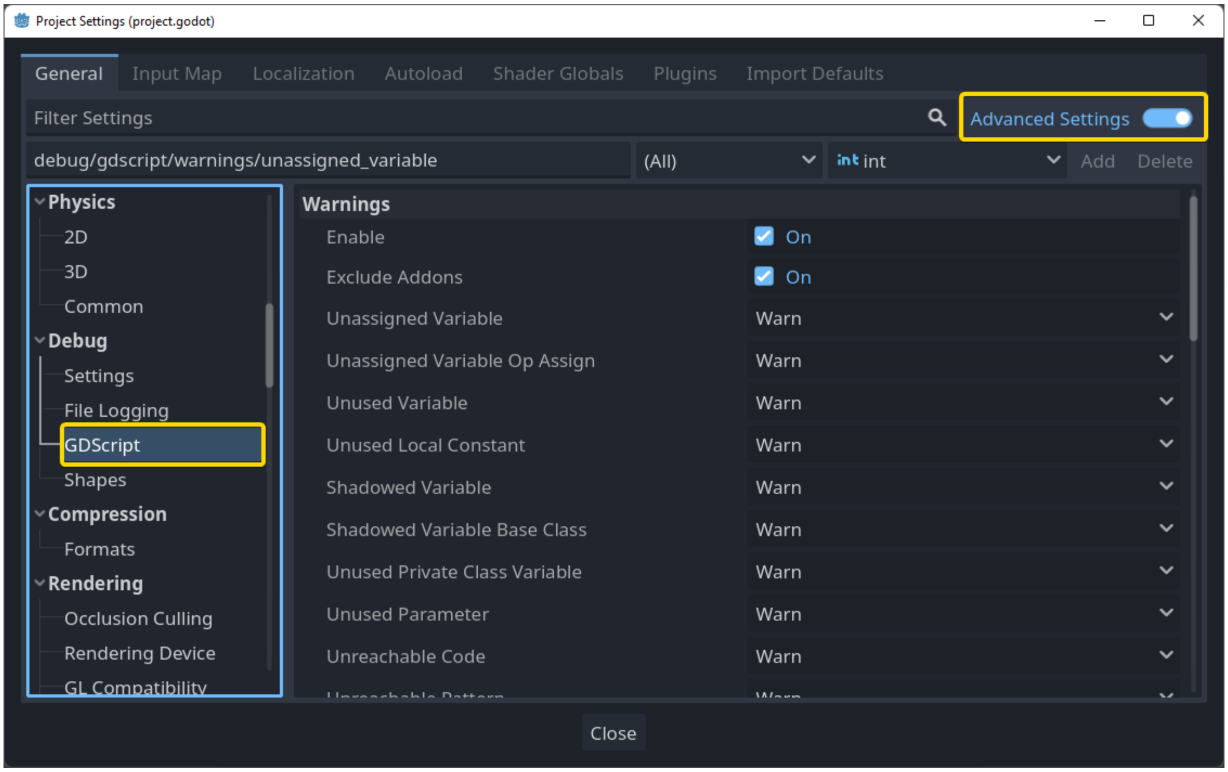
Summary

Typed GDScript is a powerful tool. It helps you write more structured code, avoid common errors, and create scalable and reliable systems. Static types improve GDScript performance and more optimizations are planned for the future.

GDScript warning system

The GDScript warning system complements static typing (but it can work without static typing too). It's here to help you avoid mistakes that are hard to spot during development, and that may lead to runtime errors.

You can configure warnings in the Project Settings under the section called **GDScript**:


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
(All)
int int
Add
Delete
Physics
2D
3D
Common
Debug
Settings
File Logging
GDScript
Shapes
Compression
Formats
Rendering
Occlusion Culling
Rendering Device
GL Compatibility
Close

debug/gdscript/warnings/unassigned_variable

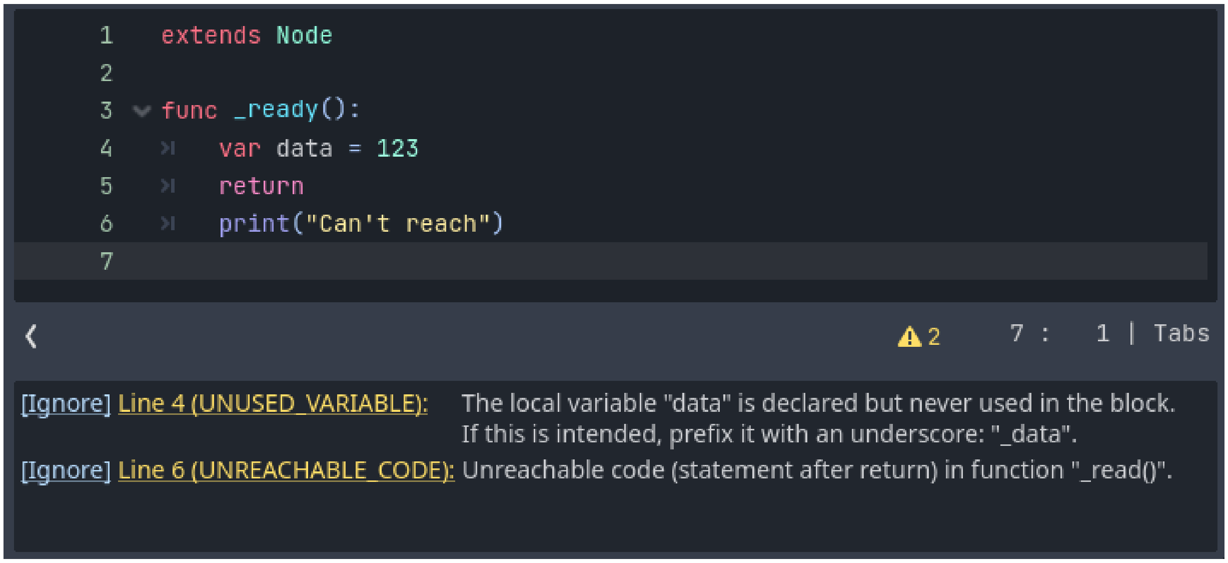
Warnings

Enable	☑ On
Exclude Addons	☑ On
Unassigned Variable	Warn
Unassigned Variable Op Assign	Warn
Unused Variable	Warn
Unused Local Constant	Warn
Shadowed Variable	Warn
Shadowed Variable Base Class	Warn
Unused Private Class Variable	Warn
Unused Parameter	Warn
Unreachable Code	Warn

Note

You must enable **Advanced Settings** in order to see the GDScript section in the sidebar. You can also search for "GDScript" when Advanced Settings is off.

You can find a list of warnings for the active GDScript file in the script editor's status bar. The example below has 2 warnings:

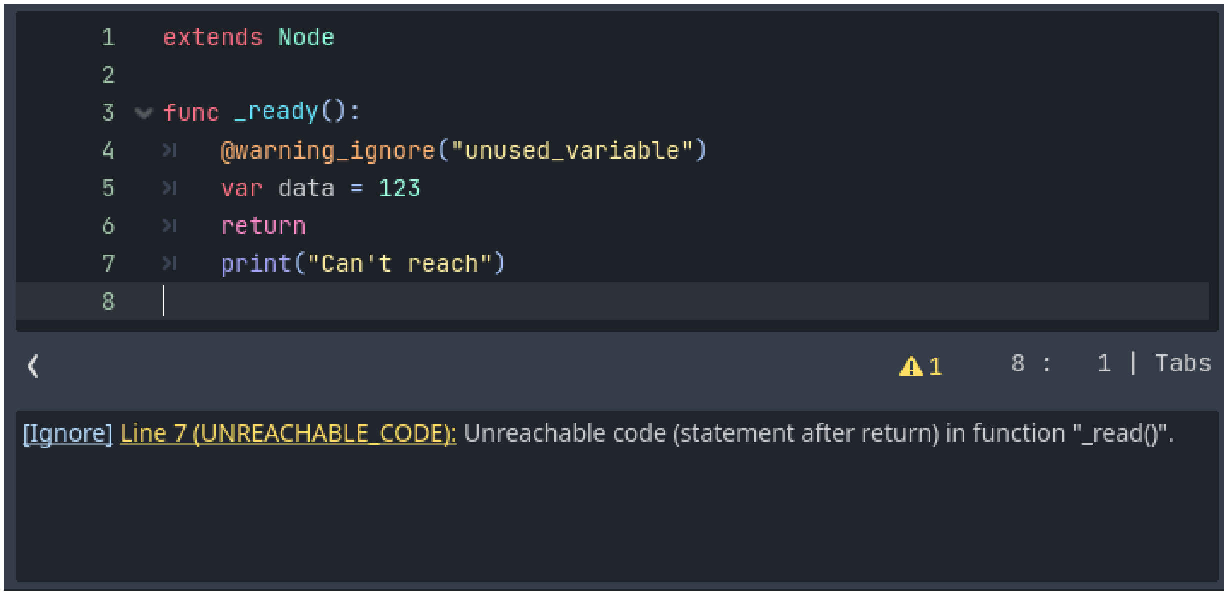


```
1   extends Node
2
3   func _ready():
4       var data = 123
5       return
6       print("Can't reach")
7
```

[Ignore] Line 4 (UNUSED_VARIABLE): The local variable "data" is declared but never used in the block. If this is intended, prefix it with an underscore: "_data".

[Ignore] Line 6 (UNREACHABLE_CODE): Unreachable code (statement after return) in function "_read()".

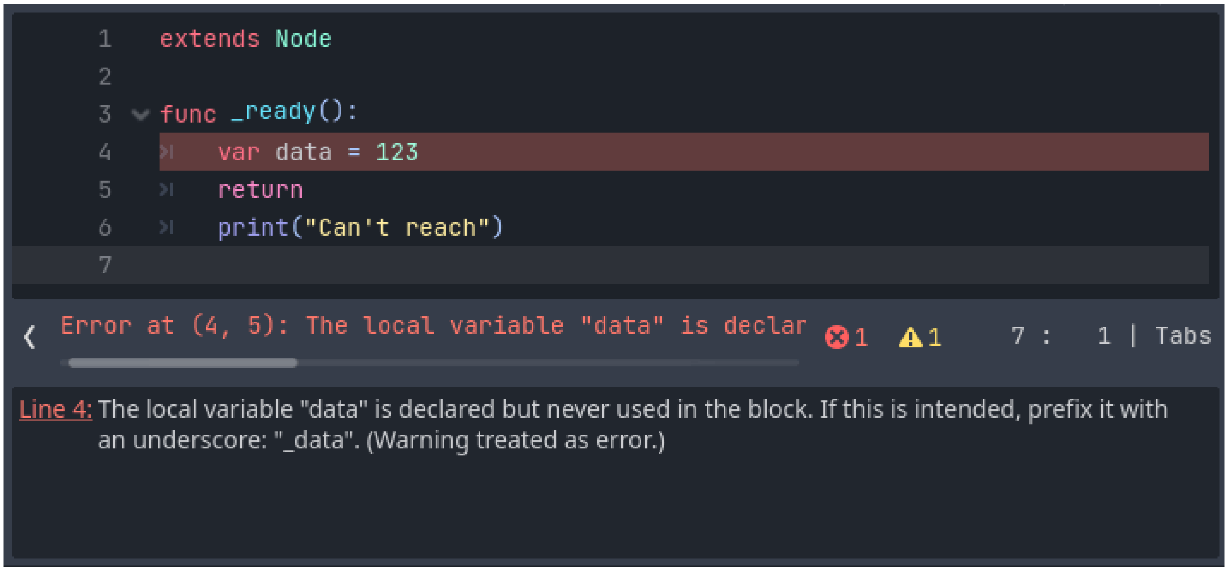
To ignore specific warnings in one file, insert an annotation of the form `@warning_ignore("warning-id")`, or click on the ignore link to the left of the warning's description. Godot will add an annotation above the corresponding line and the code won't trigger the corresponding warning anymore:



```
1   extends Node
2
3   func _ready():
4       @warning_ignore("unused_variable")
5       var data = 123
6       return
7       print("Can't reach")
8
```

[Ignore] Line 7 (UNREACHABLE_CODE): Unreachable code (statement after return) in function "_read()".

Warnings won't prevent the game from running, but you can turn them into errors if you'd like. This way your game won't compile unless you fix all warnings. Head to the `GDScript` section of the Project Settings to turn on this option to the warning that you want. Here's the same file as the previous example with the warning `unused_variable` as an error turned on:



```
1   extends Node
2
3   func _ready():
4       var data = 123
5       return
6       print("Can't reach")
7
```

```
Error at (4, 5): The local variable "data" is declar
```

Line 4: The local variable "data" is declared but never used in the block. If this is intended, prefix it with an underscore: "_data". (Warning treated as error.)

GDScript format strings

GDScript offers a feature called *format strings*, which allows reusing text templates to succinctly create different but similar strings.

Format strings are just like normal strings, except they contain certain placeholder character-sequences. These placeholders can then easily be replaced by parameters handed to the format string.

As an example, with `%s` as a placeholder, the format string `"Hello %s, how are you?"` can easily be changed to `"Hello World, how are you?"`. Notice the placeholder is in the middle of the string; modifying it without format strings could be cumbersome.

Usage in GDScript

Examine this concrete GDScript example:

```
# Define a format string with placeholder '%s'
var format_string = "We're waiting for %s."

# Using the '%' operator, the placeholder is replaced with the
desired value
var actual_string = format_string % "Godot"

print(actual_string)
# Output: "We're waiting for Godot."
```

Placeholders always start with a `%`, but the next character or characters, the *format specifier*, determines how the given value is converted to a string.

The `%s` seen in the example above is the simplest placeholder and works for most use cases: it converts the value by the same method by which an implicit String conversion or `str()` would convert it.

Strings remain unchanged, Booleans turn into either `"True"` or `"False"`, an integral or real number becomes a decimal, other types usually return their data in a human-readable string.

There is also another way to format text in GDScript, namely the `String.format()` method. It replaces all occurrences of a key in the string with the corresponding value. The method can handle arrays or dictionaries for the key/value pairs.

Arrays can be used as key, index, or mixed style (see below examples). Order only matters when the index or mixed style of Array is used.

A quick example in GDScript:

```
# Define a format string
var format_string = "We're waiting for {str}"

# Using the 'format' method, replace the 'str' placeholder
var actual_string = format_string.format({"str": "Godot"})

print(actual_string)
# Output: "We're waiting for Godot"
```

There are other format specifiers, but they are only applicable when using the `%` operator.

Multiple placeholders

Format strings may contain multiple placeholders. In such a case, the values are handed in the form of an array, one value per placeholder (unless using a format specifier with `*`, see dynamic padding):

```
var format_string = "%s was reluctant to learn %s, but now he enjoys it."
var actual_string = format_string % ["Estragon", "GDScript"]

print(actual_string)
```

```
# Output: "Estragon was reluctant to learn GDScript, but now he
enjoys it."
```

Note the values are inserted in order. Remember all placeholders must be replaced at once, so there must be an appropriate number of values.

Format specifiers

There are format specifiers other than `s` that can be used in placeholders. They consist of one or more characters. Some of them work by themselves like `s`, some appear before other characters, some only work with certain values or characters.

Placeholder types

One and only one of these must always appear as the last character in a format specifier. Apart from `s`, these require certain types of parameters.

`s`	**Simple** conversion to String by the same method as implicit String conversion.
`c`	A single **Unicode character**. Expects an unsigned 8-bit integer (0-255) for a code point or a single-character string.
`d`	A **decimal integral** number. Expects an integral or real number (will be floored).
`o`	An **octal integral** number. Expects an integral or real number (will be floored).

`x`	A **hexadecimal integral** number with **lower-case** letters. Expects an integral or real number (will be floored).
`X`	A **hexadecimal integral** number with **upper-case** letters. Expects an integral or real number (will be floored).
`f`	A **decimal real** number. Expects an integral or real number.
`v`	A **vector**. Expects any float or int-based vector object (`Vector2`, `Vector3`, `Vector4`, `Vector2i`, `Vector3i` or `Vector4i`). Will display the vector coordinates in parentheses, formatting each coordinate as if it was an `%f`, and using the same modifiers.

Placeholder modifiers

These characters appear before the above. Some of them work only under certain conditions.

`+`	In number specifiers, **show + sign** if positive.
Integer	Set **padding**. Padded with spaces or with zeroes if integer starts with `0` in an integer or real number placeholder. The leading `0` is ignored if `-` is present. When used after `.`, see `.`.
`.`	Before `f` or `v`, set **precision** to 0 decimal places. Can be followed up with numbers to change. Padded with zeroes.

`-`	**Pad to the right** rather than the left.
`*`	**Dynamic padding**, expect additional integral parameter to set padding or precision after `.`, see dynamic padding.

Padding

The `.` (*dot*), `*` (*asterisk*), `-` (*minus sign*) and digit (`0`-`9`) characters are used for padding. This allows printing several values aligned vertically as if in a column, provided a fixed-width font is used.

To pad a string to a minimum length, add an integer to the specifier:

```
print("%10d" % 12345)
# output: "     12345"
# 5 leading spaces for a total length of 10
```

If the integer starts with `0`, integral values are padded with zeroes instead of white space:

```
print("%010d" % 12345)
# output: "0000012345"
```

Precision can be specified for real numbers by adding a `.` (*dot*) with an integer following it. With no integer after `.`, a precision of 0 is used, rounding to integral value. The integer to use for padding must appear before the dot.

```
# Pad to minimum length of 10, round to 3 decimal places
print("%10.3f" % 10000.5555)
# Output: " 10000.556"
# 1 leading space
```

The `-` character will cause padding to the right rather than the left, useful for right text alignment:

```
print("%-10d" % 12345678)
# Output: "12345678  "
# 2 trailing spaces
```

Dynamic padding

By using the `*` (*asterisk*) character, the padding or precision can be set without modifying the format string. It is used in place of an integer in the format specifier. The values for padding and precision are then passed when formatting:

```
var format_string = "%*.*f"
# Pad to length of 7, round to 3 decimal places:
print(format_string % [7, 3, 8.8888])
# Output: "  8.889"
# 2 leading spaces
```

It is still possible to pad with zeroes in integer placeholders by adding `0` before `*`:

```
print("%0*d" % [2, 3])
# Output: "03"
```

Escape sequence

To insert a literal `%` character into a format string, it must be escaped to avoid reading it as a placeholder. This is done by doubling the character:

```
var health = 56
print("Remaining health: %d%%" % health)
# Output: "Remaining health: 56%"
```

Format method examples

The following are some examples of how to use the various invocations of the `String.format` method.

Type	**Style**	**Example**	**Result**
Dictionary	key	`"Hi, {name} v{version}!".format({"name":"Godette", "version":"3.0"})`	Hi, Godette v3.0!
Dictionary	index	`"Hi, {0} v{1}!".format({"0":"Godette", "1":"3.0"})`	Hi, Godette v3.0!
Dictionary	mix	`"Hi, {0} v{version}!".format({"0":"Godette", "version":"3.0"})`	Hi, Godette v3.0!
Array	key	`"Hi, {name} v{version}!".format([["version","3.0"], ["name","Godette"]])`	Hi, Godette v3.0!
Array	index	`"Hi, {0} v{1}!".format(["Godette","3.0"])`	Hi, Godette v3.0!
Array	mix	`"Hi, {name} v{0}!".format([3.0, ["name","Godette"]])`	Hi, Godette v3.0!
Array	no index	`"Hi, {} v{}!".format(["Godette", 3.0], "{}")`	Hi, Godette v3.0!

Placeholders can also be customized when using `String.format`, here's some examples of that functionality.

Type	Example	Result
Infix (default)	`"Hi, {0} v{1}".format(["Godette", "3.0"], "{_}")`	Hi, Godette v3.0
Postfix	`"Hi, 0% v1%".format(["Godette", "3.0"], "_%")`	Hi, Godette v3.0
Prefix	`"Hi, %0 v%1".format(["Godette", "3.0"], "%_")`	Hi, Godette v3.0

Combining both the `String.format` method and the `%` operator could be useful, as `String.format` does not have a way to manipulate the representation of numbers.

Example	Result
`"Hi, {0} v{version}".format({0:"Godette", "version":"%0.2f" % 3.114})`	Hi, Godette v3.11

C#/.NET

C# is a high-level programming language developed by Microsoft. Godot supports C# as an option for a scripting language, alongside Godot's own GDScript.

The standard Godot executable does not contain C# support out of the box. Instead, to enable C# support for your project you need to download a .NET version [https://godotengine.org/download/] of the editor from the Godot website.

- C# basics
- C# language features
- C# style guide
- C# diagnostics

Godot API for C#

As a general purpose game engine Godot offers some high-level features as a part of its API. Articles below explain how these features integrate into C# and how C# API may be different from GDScript.

- C# API differences to GDScript
- C# collections
- C# Variant
- C# signals
- C# exported properties
- C# global classes

C# platform support

As of Godot 4.2 projects written in C# support all desktop platforms (Windows, Linux, and macOS), as well as Android and iOS.

Android support is currently experimental and has a few limitations.

- Projects targeting Android need to target .NET 7.0 or higher. This requires modifying the *.csproj* file generated by Godot for existing projects. The following property is added to new projects created with 4.2 and can be added to existing projects:

  ```
  <TargetFramework Condition=" '$(GodotTargetPlatform)' ==
  'android' ">net7.0</TargetFramework>
  ```

- Only some Android architectures are supported: `arm64` and `x64`.

iOS support is currently experimental and has a few limitations.

- Projects targeting iOS need to target .NET 8.0 or higher. This requires modifying the *.csproj* file generated by Godot for existing projects. The following property is added to new projects created with 4.2 and can be added to existing projects:

  ```
  <TargetFramework Condition=" '$(GodotTargetPlatform)' ==
  'ios' ">net8.0</TargetFramework>
  ```

- The official export templates for the iOS simulator only supports the `x64` architecture.

- Exporting to iOS can only be done from a MacOS device.

Currently, projects written in C# cannot be exported to the web platform. To use C# on that platform, consider Godot 3 instead.

C# basics

Introduction

This page provides a brief introduction to C#, both what it is and how to use it in Godot. Afterwards, you may want to look at how to use specific features, read about the differences between the C# and the GDScript API, and (re)visit the Scripting section of the step-by-step tutorial.

C# is a high-level programming language developed by Microsoft. In Godot, it is implemented with .NET 6.0.

Attention

Projects written in C# using Godot 4 currently cannot be exported to the web platform. To use C# on the web platform, consider Godot 3 instead. Android and iOS platform support is available as of Godot 4.2, but is experimental and some limitations apply.

Note

This is **not** a full-scale tutorial on the C# language as a whole. If you aren't already familiar with its syntax or features, see the Microsoft C# guide [https://docs.microsoft.com/en-us/dotnet/csharp/index] or look for a suitable introduction elsewhere.

Prerequisites

Godot bundles the parts of .NET needed to run already compiled games. However, Godot does not bundle the tools required to build and compile games, such as MSBuild and the C# compiler. These are included in the .NET SDK, and need to be installed separately.

In summary, you must have installed the .NET SDK **and** the .NET-enabled version of Godot.

Download and install the latest stable version of the SDK from the .NET download page [https://dotnet.microsoft.com/download].

Important

Be sure to install the 64-bit version of the SDK(s) if you are using the 64-bit version of Godot.

If you are building Godot from source, make sure to follow the steps to enable .NET support in your build as outlined in the Compiling with .NET page.

Configuring an external editor

C# support in Godot's built-in script editor is minimal. Consider using an external IDE or editor, such as Visual Studio Code [https://code.visualstudio.com/] or MonoDevelop. These provide autocompletion, debugging, and other useful features for C#. To select an external editor in Godot, click on **Editor** → **Editor Settings** and scroll down to **Dotnet**. Under **Dotnet**, click on **Editor**, and select your external editor of choice. Godot currently supports the following external editors:

- Visual Studio 2022
- Visual Studio Code
- MonoDevelop

- Visual Studio for Mac
- JetBrains Rider

See the following sections for how to configure an external editor:

JetBrains Rider

After reading the "Prerequisites" section, you can download and install JetBrains Rider [https://www.jetbrains.com/rider/download].

In Godot's **Editor** → **Editor Settings** menu:

- Set **Dotnet** -> **Editor** -> **External Editor** to **JetBrains Rider**.

In Rider:

- Set **MSBuild version** to **.NET Core**.
- Install the **Godot support** plugin.

Visual Studio Code

After reading the "Prerequisites" section, you can download and install Visual Studio Code [https://code.visualstudio.com/download] (aka VS Code).

In Godot's **Editor** → **Editor Settings** menu:

- Set **Dotnet** -> **Editor** -> **External Editor** to **Visual Studio Code**.

In Visual Studio Code:

- Install the C# [https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp] extension.

To configure a project for debugging, you need a `tasks.json` and `launch.json` file in the `.vscode` folder with the necessary configuration.

Here is an example `launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Play",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${env:GODOT4}",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
        }
    ]
}
```

For this launch configuration to work, you need to either setup a GODOT4 environment variable that points to the Godot executable, or replace `program` parameter with the path to the Godot executable.

Here is an example `tasks.json`:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "command": "dotnet",
            "type": "process",
            "args": [
                "build"
            ],
            "problemMatcher": "$msCompile"
        }
    ]
}
```

Now, when you start the debugger in Visual Studio Code, your Godot project will run.

Visual Studio (Windows only)

Download and install the latest version of Visual Studio [https://visualstudio.microsoft.com/downloads/]. Visual Studio will include the required SDKs if you have the correct workloads selected, so you don't need to manually install the things listed in the "Prerequisites" section.

While installing Visual Studio, select this workload:

- .NET desktop development

In Godot's **Editor** → **Editor Settings** menu:

- Set **Dotnet** -> **Editor** -> **External Editor** to **Visual Studio**.

Note

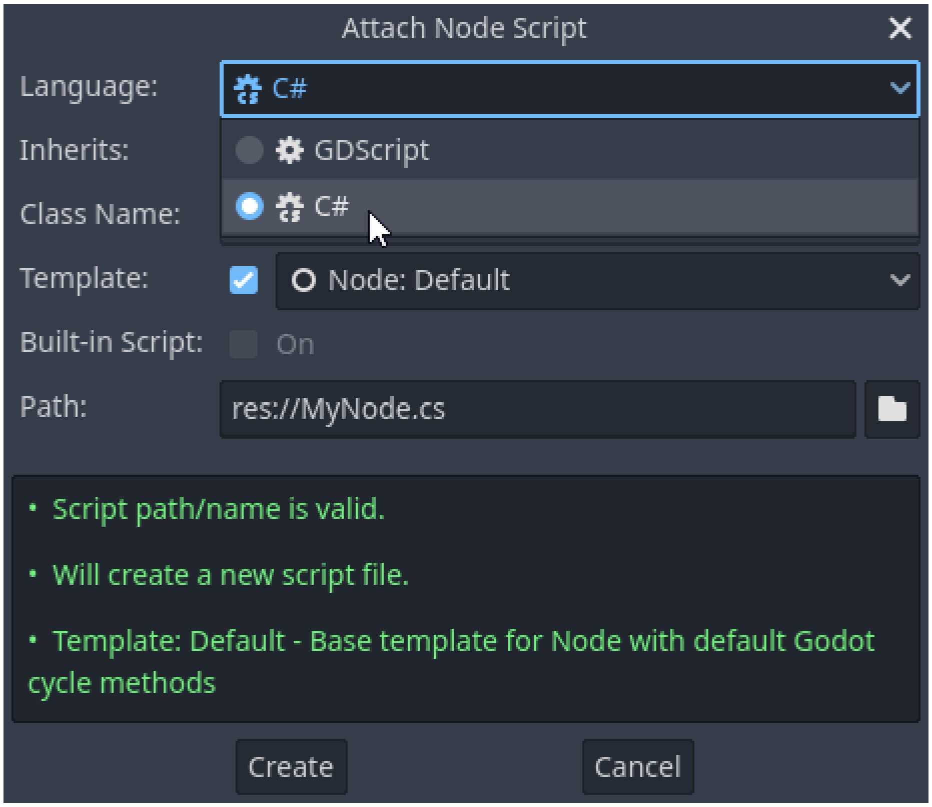
If you see an error like "Unable to find package Godot.NET.Sdk", your NuGet configuration may be incorrect and need to be fixed.

A simple way to fix the NuGet configuration file is to regenerate it. In a file explorer window, go to `%AppData%\NuGet`. Rename or delete the `NuGet.Config` file. When you build your Godot project again, the file will be automatically created with default values.

To debug your C# scripts using Visual Studio, open the .sln file that is generated after opening the first C# script in the editor. In the **Debug** menu, go to the **Debug Properties** menu item for your project. Click the **Create a new profile** button and choose **Executable**. In the **Executable** field, browse to the path of the C# version of the Godot editor, or type `%GODOT4%` if you have created an environment variable for the Godot executable path. It must be the path to the main Godot executable, not the 'console' version. For the **Working Directory**, type a single period, `.`, meaning the current directory. Also check the **Enable native code debugging** checkbox. You may now close this window, click downward arrow on the debug profile dropdown, and select your new launch profile. Hit the green start button, and your game will begin playing in debug mode.

Creating a C# script

After you successfully set up C# for Godot, you should see the following option when selecting **Attach Script** in the context menu of a node in your scene:



Language: C#

Inherits:

Class Name:

Template: ☑ Node: Default

Built-in Script: ☐ On

Path: res://MyNode.cs

- Script path/name is valid.
- Will create a new script file.
- Template: Default - Base template for Node with default Godot cycle methods

Note that while some specifics change, most concepts work the same when using C# for scripting. If you're new to Godot, you may want to follow the tutorials on Scripting languages at this point. While some documentation pages still lack C# examples, most notions can be transferred from GDScript.

Project setup and workflow

When you create the first C# script, Godot initializes the C# project files for your Godot project. This includes generating a C# solution (`.sln`) and a project file (`.csproj`), as well as some utility files and folders (`.godot/mono`). All of these but `.godot/mono` are important and should be committed to your version control system. Everything under `.godot` can be safely added to the ignore list of your VCS. When troubleshooting, it can sometimes help to delete the `.godot/mono` folder and let it regenerate.

Example

Here's a blank C# script with some comments to demonstrate how it works.

```
using Godot;

public partial class YourCustomClass : Node
{
    // Member variables here, example:
    private int _a = 2;
    private string _b = "textvar";

    public override void _Ready()
    {
        // Called every time the node is added to the scene.
        // Initialization here.
        GD.Print("Hello from C# to Godot :)");
    }
    public override void _Process(double delta)
    {
        // Called every frame. Delta is time since the last frame.
        // Update game logic here.
    }
}
```

As you can see, functions normally in global scope in GDScript like Godot's `print` function are available in the `GD` static class which is part of the `Godot` namespace. For a full list of methods in the `GD` class, see the class reference pages for @GDScript and @GlobalScope.

Note

Keep in mind that the class you wish to attach to your node should have the same name as the `.cs` file. Otherwise, you will get the following error:

"Cannot find class XXX for script res://XXX.cs"

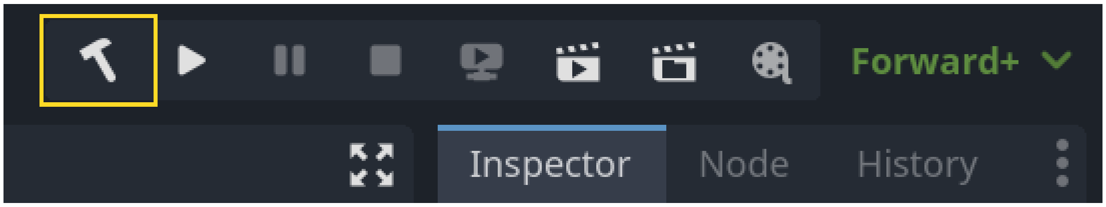
General differences between C# and GDScript

The C# API uses `PascalCase` instead of `snake_case` in GDScript/C++. Where possible, fields and getters/setters have been converted to properties. In general, the C# Godot API strives to be as idiomatic as is reasonably possible.

For more information, see the C# API differences to GDScript page.

Warning

You need to (re)build the project assemblies whenever you want to see new exported variables or signals in the editor. This build can be manually triggered by clicking the **Build** button in the top right corner of the editor.



You will also need to rebuild the project assemblies to apply changes in "tool" scripts.

Current gotchas and known issues

As C# support is quite new in Godot, there are some growing pains and things that need to be ironed out. Below is a list of the most important issues you should be aware of when diving into C# in Godot, but if in doubt, also take a look over the official issue tracker for .NET issues [https://github.com/godotengine/godot/labels/topic%3Adotnet].

- Writing editor plugins is possible, but it is currently quite convoluted.
- State is currently not saved and restored when hot-reloading, with the exception of exported variables.
- Attached C# scripts should refer to a class that has a class name that matches the file name.
- There are some methods such as `Get()`/`Set()`, `Call()`/`CallDeferred()` and signal connection method `Connect()` that rely on Godot's `snake_case` API naming conventions. So when using e.g. `CallDeferred("AddChild")`, `AddChild` will not work because the API is expecting the original `snake_case`

version `add_child`. However, you can use any custom properties or methods without this limitation. Prefer using the exposed `StringName` in the `PropertyName`, `MethodName` and `SignalName` to avoid extra `StringName` allocations and worrying about snake_case naming.

As of Godot 4.0, exporting .NET projects is supported for desktop platforms (Linux, Windows and macOS). Other platforms will gain support in future 4.x releases.

Common pitfalls

You might encounter the following error when trying to modify some values in Godot objects, e.g. when trying to change the X coordinate of a `Node2D`:

```
public partial class MyNode2D : Node2D
{
    public override void _Ready()
    {
        Position.X = 100.0f;
        // CS1612: Cannot modify the return value of
'Node2D.Position' because
        // it is not a variable.
    }
}
```

This is perfectly normal. Structs (in this example, a `Vector2`) in C# are copied on assignment, meaning that when you retrieve such an object from a property or an indexer, you get a copy of it, not the object itself. Modifying said copy without reassigning it afterwards won't achieve anything.

The workaround is simple: retrieve the entire struct, modify the value you want to modify, and reassign the property.

```
var newPosition = Position;
newPosition.X = 100.0f;
Position = newPosition;
```

Since C# 10, it is also possible to use with expressions [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/with-expression] on structs, allowing you to do the same thing in a single line.

```
Position = Position with { X = 100.0f };
```

You can read more about this error on the C# language reference [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/cs1612].

Performance of C# in Godot

According to some preliminary benchmarks [https://github.com/cart/godot3-bunnymark], the performance of C# in Godot — while generally in the same order of magnitude — is roughly **~4×** that of GDScript in some naive cases. C++ is still a little faster; the specifics are going to vary according to your use case. GDScript is likely fast enough for most general scripting workloads.

Most properties of Godot C# objects that are based on `GodotObject` (e.g. any `Node` like `Control` or `Node3D` like `Camera3D`) require native (interop) calls as they talk to Godot's C++ core. Consider assigning values of such properties into a local variable if you need to modify or read them multiple times at a single code location:

```
using Godot;

public partial class YourCustomClass : Node3D
{
    private void ExpensiveReposition()
    {
        for (var i = 0; i < 10; i++)
        {
            // Position is read and set 10 times which incurs native interop.
            // Furthermore the object is repositioned 10 times in 3D space which
            // takes additional time.
            Position += new Vector3(i, i);
        }
    }

    private void Reposition()
    {
        // A variable is used to avoid native interop for Position on every loop.
        var newPosition = Position;
        for (var i = 0; i < 10; i++)
        {
            newPosition += new Vector3(i, i);
        }
        // Setting Position only once avoids native interop and repositioning in 3D space.
        Position = newPosition;
    }
}
```

Passing raw arrays (such as `byte[]`) or `string` to Godot's C# API requires marshalling which is comparatively pricey.

The implicit conversion from `string` to `NodePath` or `StringName` incur both the native interop and marshalling costs as the `string` has to be marshalled and passed to the respective native constructor.

Using NuGet packages in Godot

NuGet [https://www.nuget.org/] packages can be installed and used with Godot, as with any C# project. Many IDEs are able to add packages directly. They can also be added manually by adding the package reference in the `.csproj` file located in the project root:

```
    <ItemGroup>
        <PackageReference Include="Newtonsoft.Json"
Version="11.0.2" />
    </ItemGroup>
    ...
</Project>
```

As of Godot 3.2.3, Godot automatically downloads and sets up newly added NuGet packages the next time it builds the project.

Profiling your C# code

The following tools may be used for performance and memory profiling of your managed code:

- JetBrains Rider with dotTrace/dotMemory plugin.
- Standalone JetBrains dotTrace/dotMemory.
- Visual Studio.

Profiling managed and unmanaged code at once is possible with both JetBrains tools and Visual Studio, but limited to Windows.

C# language features

This page provides an overview of the commonly used features of both C# and Godot and how they are used together.

Type conversion and casting

C# is a statically typed language. Therefore, you can't do the following:

```
var mySprite = GetNode("MySprite");
mySprite.SetFrame(0);
```

The method `GetNode()` returns a `Node` instance. You must explicitly convert it to the desired derived type, `Sprite2D` in this case.

For this, you have various options in C#.

Casting and Type Checking

Throws `InvalidCastException` if the returned node cannot be cast to Sprite2D. You would use it instead of the `as` operator if you are pretty sure it won't fail.

```
Sprite2D mySprite = (Sprite2D)GetNode("MySprite");
mySprite.SetFrame(0);
```

Using the AS operator

The `as` operator returns `null` if the node cannot be cast to Sprite2D, and for that reason, it cannot be used with value types.

```
Sprite2D mySprite = GetNode("MySprite") as Sprite2D;
// Only call SetFrame() if mySprite is not null
```

```
mySprite?.SetFrame(0);
```

Using the generic methods

Generic methods are also provided to make this type conversion transparent.

`GetNode<T>()` casts the node before returning it. It will throw an `InvalidCastException` if the node cannot be cast to the desired type.

```
Sprite2D mySprite = GetNode<Sprite2D>("MySprite");
mySprite.SetFrame(0);
```

`GetNodeOrNull<T>()` uses the `as` operator and will return `null` if the node cannot be cast to the desired type.

```
Sprite2D mySprite = GetNodeOrNull<Sprite2D>("MySprite");
// Only call SetFrame() if mySprite is not null
mySprite?.SetFrame(0);
```

Type checking using the IS operator

To check if the node can be cast to Sprite2D, you can use the `is` operator. The `is` operator returns false if the node cannot be cast to Sprite2D, otherwise it returns true. Note that when the `is` operator is used against `null` the result is always going to be `false`.

```
if (GetNode("MySprite") is Sprite2D)
{
    // Yup, it's a Sprite2D!
}

if (null is Sprite2D)
{
    // This block can never happen.
}
```

You can also declare a new variable to conditionally store the result of the cast if the `is` operator returns `true`.

```
if (GetNode("MySprite") is Sprite2D mySprite)
{
    // The mySprite variable only exists inside this block, and
it's never null.
    mySprite.SetFrame(0);
}
```

For more advanced type checking, you can look into Pattern Matching [https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching].

Preprocessor defines

Godot has a set of defines that allow you to change your C# code depending on the environment you are compiling to.

Examples

For example, you can change code based on the platform:

```
    public override void _Ready()
    {
#if (GODOT_32 || GODOT_MOBILE || GODOT_WEB)
        // Use simple objects when running on less powerful
systems.
        SpawnSimpleObjects();
#else
        SpawnComplexObjects();
#endif
    }
```

Or you can detect which engine your code is in, useful for making cross-engine libraries:

```
    public void MyPlatformPrinter()
    {
#if GODOT
        GD.Print("This is Godot.");
#elif UNITY_5_3_OR_NEWER
```

```
        print("This is Unity.");
#else
        throw new NotSupportedException("Only Godot and Unity
are supported.");
#endif
    }
```

Or you can write scripts that target multiple Godot versions and take advantage of features that are only available on some of those versions:

```
    public void UseCoolFeature()
    {
#if GODOT4_3_OR_GREATER || GODOT4_2_2_OR_GREATER
        // Use CoolFeature, that was added to Godot in 4.3 and
cherry-picked into 4.2.2, here.
#else
        // Use a workaround for the absence of CoolFeature
here.
#endif
    }
```

Full list of defines

- `GODOT` is always defined for Godot projects.
- `TOOLS` is defined when building with the Debug configuration (editor and editor player).
- `GODOT_REAL_T_IS_DOUBLE` is defined when the `GodotFloat64` property is set to `true`.
- One of `GODOT_64` or `GODOT_32` is defined depending on if the architecture is 64-bit or 32-bit.
- One of `GODOT_LINUXBSD`, `GODOT_WINDOWS`, `GODOT_OSX`, `GODOT_ANDROID`, `GODOT_IOS`, `GODOT_WEB` depending on the OS. These names may change in the future. These are created from

the `get_name()` method of the OS singleton, but not every possible OS the method returns is an OS that Godot with .NET runs on.

- `GODOTX`, `GODOTX_Y`, `GODOTX_Y_Z`, `GODOTx_OR_GREATER`, `GODOTX_y_OR_GREATER`, and `GODOTX_Y_z_OR_GREATER`, where `X`, `Y`, and `Z` are replaced by the current major, minor and patch version of Godot. `x`, `y`, and `z` are replaced by all values from 0 to the current version number for that component.

  Note

  These defines were first added in Godot 4.0.4 and 4.1. Version defines for prior versions do not exist, regardless of the current Godot version.

  For example: Godot 4.0.5 defines `GODOT4`, `GODOT4_OR_GREATER`, `GODOT4_0`, `GODOT4_0_OR_GREATER`, `GODOT4_0_5`, `GODOT4_0_4_OR_GREATER`, and `GODOT4_0_5_OR_GREATER`. Godot 4.3.2 defines `GODOT4`, `GODOT4_OR_GREATER`, `GODOT4_3`, `GODOT4_0_OR_GREATER`, `GODOT4_1_OR_GREATER`, `GODOT4_2_OR_GREATER`, `GODOT4_3_OR_GREATER`, `GODOT4_3_2`, `GODOT4_3_0_OR_GREATER`, `GODOT4_3_1_OR_GREATER`, and `GODOT4_3_2_OR_GREATER`.

When **exporting**, the following may also be defined depending on the export features:

- One of `GODOT_PC`, `GODOT_MOBILE`, or `GODOT_WEB` depending on the platform type.
- One of `GODOT_WINDOWS`, `GODOT_LINUXBSD`, `GODOT_MACOS`, `GODOT_ANDROID`, `GODOT_IOS`, or `GODOT_WEB` depending on the platform.

To see an example project, see the OS testing demo: https://github.com/godotengine/godot-demo-projects/tree/master/misc/os_test

C# style guide

Having well-defined and consistent coding conventions is important for every project, and Godot is no exception to this rule.

This page contains a coding style guide, which is followed by developers of and contributors to Godot itself. As such, it is mainly intended for those who want to contribute to the project, but since the conventions and guidelines mentioned in this article are those most widely adopted by the users of the language, we encourage you to do the same, especially if you do not have such a guide yet.

Note

This article is by no means an exhaustive guide on how to follow the standard coding conventions or best practices. If you feel unsure of an aspect which is not covered here, please refer to more comprehensive documentation, such as C# Coding Conventions [https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions] or Framework Design Guidelines [https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines].

Language specification

Godot currently uses **C# version 10.0** in its engine and example source code, as this is the version supported by .NET 6.0 (the current baseline requirement). So, before we move to a newer version, care must be taken to avoid mixing language features only available in C# 11.0 or later.

For detailed information on C# features in different versions, please see What's New in C# [https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/].

Formatting

General guidelines

- Use line feed (**LF**) characters to break lines, not CRLF or CR.
- Use one line feed character at the end of each file, except for *csproj* files.
- Use **UTF-8** encoding without a byte order mark [https://en.wikipedia.org/wiki/Byte_order_mark].
- Use **4 spaces** instead of tabs for indentation (which is referred to as "soft tabs").
- Consider breaking a line into several if it's longer than 100 characters.

Line breaks and blank lines

For a general indentation rule, follow the "Allman Style" [https://en.wikipedia.org/wiki/Indentation_style#Allman_style] which recommends placing the brace associated with a control statement on the next line, indented to the same level:

```
// Use this style:
if (x > 0)
{
    DoSomething();
}

// NOT this:
if (x > 0) {
    DoSomething();
}
```

However, you may choose to omit line breaks inside brackets:

- For simple property accessors.
- For simple object, array, or collection initializers.
- For abstract auto property, indexer, or event declarations.

```
// You may put the brackets in a single line in following cases:
public interface MyInterface
{
    int MyProperty { get; set; }
}

public class MyClass : ParentClass
{
    public int Value
    {
        get { return 0; }
        set
        {
            ArrayValue = new [] {value};
        }
    }
}
```

Insert a blank line:

- After a list of `using` statements.
- Between method, properties, and inner type declarations.
- At the end of each file.

Field and constant declarations can be grouped together according to relevance. In that case, consider inserting a blank line between the groups for easier reading.

Avoid inserting a blank line:

- After `{`, the opening brace.
- Before `}`, the closing brace.
- After a comment block or a single-line comment.

- Adjacent to another blank line.

```
using System;
using Godot;
                                          // Blank line after
`using` list.
public class MyClass
{                                         // No blank line
after `{`.
    public enum MyEnum
    {
        Value,
        AnotherValue                      // No blank line
before `}`.
    }
                                          // Blank line around
inner types.
    public const int SomeConstant = 1;
    public const int AnotherConstant = 2;

    private Vector3 _x;                  // Related constants
or fields can be
    private Vector3 _y;                  // grouped together.

    private float _width;
    private float _height;

    public int MyProperty { get; set; }
                                          // Blank line around
properties.
    public void MyMethod()
    {
        // Some comment.
        AnotherMethod();                  // No blank line
after a comment.
    }
                                          // Blank line around
methods.
    public void AnotherMethod()
    {
    }
}
```

Using spaces

Insert a space:

- Around a binary and ternary operator.
- Between an opening parenthesis and `if`, `for`, `foreach`, `catch`, `while`, `lock` or `using` keywords.
- Before and within a single line accessor block.
- Between accessors in a single line accessor block.
- After a comma which is not at the end of a line.
- After a semicolon in a `for` statement.
- After a colon in a single line `case` statement.
- Around a colon in a type declaration.
- Around a lambda arrow.
- After a single-line comment symbol (`//`), and before it if used at the end of a line.
- After the opening brace, and before the closing brace in a single line initializer.

Do not use a space:

- After type cast parentheses.

The following example shows a proper use of spaces, according to some of the above mentioned conventions:

```
public class MyClass<A, B> : Parent<A, B>
{
    public float MyProperty { get; set; }

    public float AnotherProperty
    {
        get { return MyProperty; }
    }

    public void MyMethod()
    {
        int[] values = { 1, 2, 3, 4 };
        int sum = 0;

        // Single line comment.
        for (int i = 0; i < values.Length; i++)
        {
            switch (i)
            {
                case 3: return;
                default:
                    sum += i > 2 ? 0 : 1;
                    break;
            }
        }

        i += (int)MyProperty; // No space after a type cast.
    }
}
```

Naming conventions

Use **PascalCase** for all namespaces, type names and member level identifiers (i.e. methods, properties, constants, events), except for private fields:

```
namespace ExampleProject
{
    public class PlayerCharacter
    {
        public const float DefaultSpeed = 10f;

        public float CurrentSpeed { get; set; }

        protected int HitPoints;

        private void CalculateWeaponDamage()
        {
        }
    }
}
```

Use **camelCase** for all other identifiers (i.e. local variables, method arguments), and use an underscore (_) as a prefix for private fields (but not for methods or properties, as explained above):

```
private Vector3 _aimingAt; // Use a `_` prefix for private fields.

private void Attack(float attackStrength)
{
    Enemy targetFound = FindTarget(_aimingAt);

    targetFound?.Hit(attackStrength);
}
```

There's an exception with acronyms which consist of two letters, like `UI`, which should be written in uppercase letters where PascalCase would be expected, and in lowercase letters otherwise.

Note that `id` is **not** an acronym, so it should be treated as a normal identifier:

```
public string Id { get; }

public UIManager UI
{
    get { return uiManager; }
}
```

It is generally discouraged to use a type name as a prefix of an identifier, like `string strText` or `float fPower`, for example. An exception is made, however, for interfaces, which **should**, in fact, have an uppercase letter `I` prefixed to their names, like `IInventoryHolder` or `IDamageable`.

Lastly, consider choosing descriptive names and do not try to shorten them too much if it affects readability.

For instance, if you want to write code to find a nearby enemy and hit it with a weapon, prefer:

```
FindNearbyEnemy()?.Damage(weaponDamage);
```

Rather than:

```
FindNode()?.Change(wpnDmg);
```

Member variables

Don't declare member variables if they are only used locally in a method, as it makes the code more difficult to follow. Instead, declare them as local variables in the method's body.

Local variables

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Implicitly typed local variables

Consider using implicitly typing (`var`) for declaration of a local variable, but do so **only when the type is evident** from the right side of the assignment:

```
// You can use `var` for these cases:

var direction = new Vector2(1, 0);

var value = (int)speed;

var text = "Some value";

for (var i = 0; i < 10; i++)
{
}
```

```
// But not for these:

var value = GetValue();

var velocity = direction * 1.5;

// It's generally a better idea to use explicit typing for
numeric values, especially with
// the existence of the `real_t` alias in Godot, which can
either be double or float
// depending on the build configuration.

var value = 1.5;
```

Other considerations

- Use explicit access modifiers.
- Use properties instead of non-private fields.
- Use modifiers in this order: `public`/`protected`/`private`/`internal`/`virtual`/`override`/`abstract`/`new`/`static`/`readonly`.
- Avoid using fully-qualified names or `this.` prefix for members when it's not necessary.
- Remove unused `using` statements and unnecessary parentheses.
- Consider omitting the default initial value for a type.
- Consider using null-conditional operators or type initializers to make the code more compact.
- Use safe cast when there is a possibility of the value being a different type, and use direct cast otherwise.

C# diagnostics

Godot includes analyzers that inspect your C# source code to check for invalid or unsupported code and let you know that something is wrong during build time.

Rules

- GD0001: Missing partial modifier on declaration of type that derives from GodotObject
- GD0002: Missing partial modifier on declaration of type which contains nested classes that derive from GodotObject
- GD0003: Found multiple classes with the same name in the same script file
- GD0101: The exported member is static
- GD0102: The type of the exported member is not supported
- GD0103: The exported member is read-only
- GD0104: The exported property is write-only
- GD0105: The exported property is an indexer
- GD0106: The exported property is an explicit interface implementation
- GD0107: Types not derived from Node should not export Node members
- GD0201: The name of the delegate must end with 'EventHandler'
- GD0202: The parameter of the delegate signature of the signal is not supported
- GD0203: The delegate signature of the signal must return void
- GD0301: The generic type argument must be a Variant compatible type
- GD0302: The generic type parameter must be annotated with the '[MustBeVariant]' attribute

- GD0303: The parent symbol of a type argument that must be Variant compatible was not handled
- GD0401: The class must derive from Godot.GodotObject or a derived class
- GD0402: The class must not be generic

GD0001: Missing partial modifier on declaration of type that derives from GodotObject

Rule ID	GD0001
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

A type that derives from `GodotObject` is not declared partial.

Rule description

Godot source generators add generated code to user-defined types to implement the integration with the engine. Source generators can't add generated code to types that aren't declared partial.

```
// The source generators can't enhance this type to work with
Godot.
public class InvalidNode : Node { }

// The source generators can enhance this type to work with
Godot.
public partial class ValidNode { }
```

How to fix violations

To fix a violation of this rule, add the `partial` keyword to the type declaration.

When to suppress warnings

Do not suppress a warning from this rule. Types that derive from `GodotObject` but aren't partial can't be enhanced by the source generators, resulting in unexpected runtime errors.

GD0002: Missing partial modifier on declaration of type which contains nested classes that derive from GodotObject

Rule ID	GD0002
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

A type that derives from `GodotObject` is contained in a non-partial type declaration.

Rule description

Godot source generators add generated code to user-defined types to implement the integration with the engine. Source generators can't add generated code to types that aren't declared partial.

```
public class InvalidParentType
{
    // MyNode is contained in a non-partial type so the source
generators
    // can't enhance this type to work with Godot.
    public partial class MyNode : Node { }
}

public partial class ValidParentType
{
    // MyNode is contained in a partial type so the source
generators
    // can enhance this type to work with Godot.
    public partial class MyNode : Node { }
}
```

How to fix violations

To fix a violation of this rule, add the `partial` keyword to the type declaration.

When to suppress warnings

Do not suppress a warning from this rule. Types that derive from `GodotObject` but aren't partial can't be enhanced by the source generators, resulting in unexpected runtime errors.

GD0003: Found multiple classes with the same name in the same script file

Rule ID	GD0003
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

A script file contains multiple types that derives from `GodotObject` with a name that matches the script file. Only one type in the script file should match the file name.

Rule description

Godot requires scripts to have a unique path so every type must be defined on its own file and the type name must match the file name.

```
public partial class MyNode : Node { }

namespace DifferentNamespace
{
    // Invalid because there's already a type with the name MyNode in this file.
    public partial class MyNode : Node { }
}

// Invalid because there's already a type with the name MyNode in this file.
public partial class MyNode<T> : Node { }
```

How to fix violations

To fix a violation of this rule, move each type declaration to a different file.

When to suppress warnings

Do not suppress a warning from this rule. Types that derive from `GodotObject` must have a unique path otherwise the engine can't load the script by path, resulting in unexpected runtime errors.

GD0101: The exported member is static

Rule ID	GD0101
Category	Usage
Fix is breaking or non-breaking	Breaking - If the `static` keyword is removed Non-breaking - If the `[Export]` attribute is removed
Enabled by default	Yes

Cause

A static member is annotated with the `[Export]` attribute. Static members can't be exported.

Rule description

Godot doesn't allow exporting static members.

```csharp
// Static members can't be exported.
[Export]
public static int InvalidProperty { get; set; }
```

```
// Instance members can be exported.
[Export]
public int ValidProperty { get; set; }
```

How to fix violations

To fix a violation of this rule, remove the `[Export]` attribute or remove the `static` keyword.

When to suppress warnings

Do not suppress a warning from this rule. Static members can't be exported so they will be ignored by Godot, resulting in runtime errors.

GD0102: The type of the exported member is not supported

Rule ID	GD0102
Category	Usage
Fix is breaking or non-breaking	Breaking - If the member type is changed Non-breaking - If the `[Export]` attribute is removed
Enabled by default	Yes

Cause

An unsupported type is specified for a member annotated with the `[Export]` attribute when a Variant-compatible type is expected.

Rule description

Every exported member must be Variant-compatible so it can be marshalled by the engine.

```csharp
class SomeType { }

// SomeType is not a valid member type because it doesn't derive from GodotObject,
// so it's not compatible with Variant.
[Export]
public SomeType InvalidProperty { get; set; }

// System.Int32 is a valid type because it's compatible with Variant.
[Export]
public int ValidProperty { get; set; }
```

How to fix violations

To fix a violation of this rule, change the member's type to be Variant-compatible or remove the `[Export]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Members with types that can't be marshalled will result in runtime errors.

GD0103: The exported member is read-only

Rule ID	GD0103
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

A read-only member is annotated with the `[Export]` attribute. Read-only members can't be exported.

Rule description

Godot doesn't allow exporting read-only members.

```
// Read-only fields can't be exported.
[Export]
public readonly int invalidField;

// This field can be exported because it's not declared
'readonly'.
```

```
[Export]
public int validField;

// Read-only properties can't be exported.
[Export]
public int InvalidProperty { get; }

// This property can be exported because it has both a getter
and a setter.
[Export]
public int ValidProperty { get; set; }
```

How to fix violations

To fix a violation of this rule for fields, remove the `readonly` keyword or remove the `[Export]` attribute.

To fix a violation of this rule for properties, make sure the property declares both a getter and a setter, or remove the `[Export]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Read-only members can't be exported so they will be ignored by Godot, resulting in runtime errors.

GD0104: The exported property is write-only

Rule ID	GD0104
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

A write-only property is annotated with the `[Export]` attribute. Write-only properties can't be exported.

Rule description

Godot doesn't allow exporting write-only properties.

```
private int _backingField;

// Write-only properties can't be exported.
[Export]
public int InvalidProperty { set => _backingField = value; }
```

```
// This property can be exported because it has both a getter
and a setter.
[Export]
public int ValidProperty { get; set; }
```

How to fix violations

To fix a violation of this rule, make sure the property declares both a getter and a setter, or remove the `[Export]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Write-only members can't be exported so they will be ignored by Godot, resulting in runtime errors.

GD0105: The exported property is an indexer

Rule ID	GD0105
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

An indexer is annotated with the `[Export]` attribute. Indexers can't be exported.

Rule description

Godot doesn't allow exporting indexer properties.

```
private int[] _backingField;

// Indexers can't be exported.
[Export]
public int this[int index]
{
    get => _backingField[index];
    set => _backingField[index] = value;
}
```

How to fix violations

To fix a violation of this rule, remove the `[Export]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Indexers can't be exported so they will be ignored by Godot, resulting in runtime errors.

GD0106: The exported property is an explicit interface implementation

Rule ID	GD0106
Category	Usage
Fix is breaking or non-breaking	Non-breaking
Enabled by default	Yes

Cause

An explicit interface property implementation is annotated with the `[Export]` attribute. Properties that implement an interface explicitly can't be exported.

Rule description

Godot doesn't allow exporting explicit interface property implementations. When an interface member is implemented explicitly, the member is hidden and consumers can't access them unless the type is converted to the interface first. Explicitly implemented members can also share the same name of other members in the type, so it could create naming conflicts with other exported members.

```
public interface MyInterface
{
    public int MyProperty { get; set; }
}

public class MyNode1 : Node, MyInterface
{
    // The property can be exported because it implements the
interface implicitly.
    [Export]
    public int MyProperty { get; set; }
}

public class MyNode2 : Node, MyInterface
{
    // The property can't be exported because it implements the
interface explicitly.
    [Export]
    int MyInterface.MyProperty { get; set; }
}
```

How to fix violations

To fix a violation of this rule, implement the interface implicitly or remove the `[Export]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Explicit interface property implementations can't be exported so they will be ignored by Godot, resulting in runtime errors.

GD0107: Types not derived from Node should not export Node members

Rule ID	GD0107
Category	Usage
Fix is breaking or non-breaking	Breaking
Enabled by default	Yes

Cause

A type that doesn't derive from `Node` contains an exported field or property of a type that derives from `Node`.

Rule description

Exported nodes are serialized as `NodePath`. Only types derived from `Node` are able to get the node instance from the `NodePath`.

How to fix violations

To fix a violation of this rule, avoid exporting `Node` members on a type that doesn't derive from `Node`, or consider exporting a `NodePath`.

When to suppress warnings

Do not suppress a warning from this rule. Types that don't derive from `Node` will be unable to retrieve the right node instance for exported `Node` members, resulting in unexpected runtime errors.

GD0201: The name of the delegate must end with 'EventHandler'

Rule ID	GD0201
Category	Usage
Fix is breaking or non-breaking	Breaking
Enabled by default	Yes

Cause

A delegate annotated with the `[Signal]` attribute has a name that doesn't end with 'EventHandler'.

Rule description

Godot source generators will generate C# events using the name of the delegate with the 'EventHandler' suffix removed. Adding the 'EventHandler' suffix to the name of delegates used in events is a .NET naming convention [https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-classes-structs-and-interfaces#names-of-common-types].

Using a suffix for the delegate allows the generated event to use the name without the suffix avoiding a naming conflict.

```
// This delegate is invalid since the name doesn't end with
'EventHandler'.
[Signal]
public void InvalidSignal();

// This delegate is valid since the name ends with
'EventHandler'.
[Signal]
public void ValidSignalEventHandler();
```

Take a look at the C# signals documentation for more information about how to declare and use signals.

How to fix violations

To fix a violation of this rule, add 'EventHandler' to the end of the delegate name.

When to suppress warnings

Do not suppress a warning from this rule. Signal delegates without the suffix will be ignored by the source generator, so the signal won't be registered.

GD0202: The parameter of the delegate signature of the signal is not supported

Rule ID	GD0202
Category	Usage
Fix is breaking or non-breaking	Breaking - If the parameter type is changed Non-breaking - If the `[Signal]` attribute is removed
Enabled by default	Yes

Cause

An unsupported type is specified for a parameter of a delegate annotated with the `[Signal]` attribute when a Variant-compatible type is expected.

Rule description

Every signal parameter must be Variant-compatible so it can be marshalled when emitting the signal and invoking the callbacks.

```
class SomeType { }

// SomeType is not a valid parameter type because it doesn't
derive from GodotObject,
// so it's not compatible with Variant.
public void InvalidSignalEventHandler(SomeType someType);

// System.Int32 is a valid type because it's compatible with
Variant.
public void ValidSignalEventHandler(int someInt);
```

Take a look at the C# signals documentation for more information about how to declare and use signals.

How to fix violations

To fix a violation of this rule, change the parameter type to be Variant-compatible or remove the `[Signal]` attribute from the delegate. Note that removing the attribute will mean the signal is not registered.

Tip

If the signal doesn't need to interact with Godot, consider using C# events [https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/] directly. Pure C# events allow you to use any C# type for its parameters.

When to suppress warnings

Do not suppress a warning from this rule. Signal delegates with parameters that can't be marshalled will result in runtime errors when emitting the signal or invoking the callbacks.

GD0203: The delegate signature of the signal must return void

Rule ID	GD0203
Category	Usage
Fix is breaking or non-breaking	Breaking - If the return type is changed Non-breaking - If the `[Signal]` attribute is removed
Enabled by default	Yes

Cause

A delegate annotated with the `[Signal]` attribute has a return type when `void` was expected.

Rule description

Every signal must return `void`. There can be multiple callbacks registered for each signal, if signal callbacks could return something it wouldn't be possible to determine which of the returned values to use.

```
// This signal delegate is invalid because it doesn't return
void.
public int InvalidSignalEventHandler();

// This signal delegate is valid because it returns void.
public void ValidSignalEventHandler();
```

Take a look at the C# signals documentation for more information about how to declare and use signals.

How to fix violations

To fix a violation of this rule, change the delegate to return `void` or remove the `[Signal]` attribute from the delegate. Note that removing the attribute will mean the signal is not registered.

Tip

If the signal doesn't need to interact with Godot, consider using C# events [https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/] directly. Pure C# events allow you to use any C# type for its parameters.

When to suppress warnings

Do not suppress a warning from this rule. Signal delegates that return something will result in unexpected runtime errors.

GD0301: The generic type argument must be a Variant compatible type

Rule ID	GD0301
Category	Usage
Fix is breaking or non-breaking	Breaking
Enabled by default	Yes

Cause

An unsupported type is specified for a generic type argument when a Variant-compatible type is expected.

Rule description

When a generic type parameter is annotated with the `[MustBeVariant]` attribute, the generic type is required to be a Variant-compatible type. For example, the generic `Godot.Collections.Array<T>` type only supports items of a type that can be converted to Variant.

```
class SomeType { }

// SomeType is not a valid type because it doesn't derive from
GodotObject,
// so it's not compatible with Variant.
var invalidArray = new Godot.Collections.Array<SomeType>();

// System.Int32 is a valid type because it's compatible with
Variant.
var validArray = new Godot.Collections.Array<int>();
```

How to fix violations

To fix a violation of this rule, change the generic type argument to be a Variant-compatible type or use a different API that doesn't require the generic type argument to be a Variant-compatible type.

When to suppress warnings

Do not suppress a warning from this rule. API that contains generic type arguments annotated with the `[MustBeVariant]` attribute usually has this requirement because the values will be passed to the engine, if the type can't be marshalled it will result in runtime errors.

GD0302: The generic type parameter must be annotated with the '[MustBeVariant]' attribute

Rule ID	GD0302
Category	Usage
Fix is breaking or non-breaking	Breaking
Enabled by default	Yes

Cause

A generic type is specified for a generic type argument when a Variant-compatible type is expected, but the specified generic type is not annotated with the `[MustBeVariant]` attribute.

Rule description

When a generic type parameter is annotated with the `[MustBeVariant]` attribute, the generic type is required to be a

Variant-compatible type. When the type used is also a generic type, this generic type must be annotated with the `[MustBeVariant]` attribute as well. For example, the generic `Godot.Collections.Array<T>` type only supports items of a type that can be converted to Variant, a generic type can be specified if it's properly annotated.

```
public void Method1<T>()
{
    // T is not valid here because it may not a Variant-
compatible type.
    var invalidArray = new Godot.Collections.Array<T>();
}

public void Method2<[MustBeVariant] T>()
{
    // T is guaranteed to be a Variant-compatible type because
it's annotated
    // with the [MustBeVariant] attribute, so it can be used
here.
    var validArray = new Godot.Collections.Array<T>();
}
```

How to fix violations

To fix a violation of this rule, add the `[MustBeVariant]` attribute to the generic type that is used as a generic type argument that must be Variant-compatible.

When to suppress warnings

Do not suppress a warning from this rule. API that contains generic type arguments annotated with the `[MustBeVariant]` attribute usually has this requirement because the values will be passed to the engine, if the type can't be marshalled it will result in runtime errors.

GD0303: The parent symbol of a type argument that must be Variant compatible was not handled

Rule ID	GD0303
Category	Usage
Fix is breaking or non-breaking	Not fixable
Enabled by default	Yes

Cause

This is a bug in the engine and must be reported.

Rule description

The `MustBeVariantAnalyzer` has found an unhandled case in the user source code. Please, open an issue [https://github.com/godotengine/godot/issues] and attach a minimal reproduction project so it can be fixed.

How to fix violations

Violations of this rule can't be fixed.

When to suppress warnings

Suppressing a warning from this rule may result in unexpected errors, since the case found by the analyzer may need to be handled by the user to prevent types that are not Variant-compatible from reaching the engine. Attempting to marshal incompatible types will result in runtime errors.

GD0401: The class must derive from Godot.GodotObject or a derived class

Rule ID	GD0401
Category	Usage
Fix is breaking or non-breaking	Breaking - If changing the inheritance chain Non-breaking - If removing the `[GlobalClass]` attribute
Enabled by default	Yes

Cause

A type annotated with the `[GlobalClass]` attribute does not derive from `GodotObject`.

Rule description

The `[GlobalClass]` has no effect for types that don't derive from `GodotObject`. Every global class must ultimately derive from `GodotObject` so it can be marshalled.

```
// This type is not registered as a global class because it doesn't derive from GodotObject.
[GlobalClass]
class SomeType { }

// This type is a global class because it derives from Godot.Node
// which ultimately derives from GodotObject.
[GlobalClass]
class MyNode : Node { }

// This type is a global class because it derives from Godot.Resource
// which ultimately derives from GodotObject.
[GlobalClass]
class MyResource : Resource { }
```

How to fix violations

To fix a violation of this rule, change the type to derive from `GodotObject` or remove the `[GlobalClass]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Adding the `[GlobalClass]` to a type that doesn't derive from `GodotObject` is an easy mistake to make and this warning helps users realize that it may result in unexpected errors.

GD0402: The class must not be generic

Rule ID	GD0402
Category	Usage
Fix is breaking or non-breaking	Breaking
Enabled by default	Yes

Cause

A generic type is annotated with the `[GlobalClass]` attribute.

Rule description

The Godot editor assumes every global class is instantiable, but generic types can't be instantiated because the type parameters are unbound.

```
// This type is a valid global class because it's not generic.
[GlobalClass]
class SomeType : Node { }

// This type is not a valid global class because it's generic.
```

```
[GlobalClass]
class SomeGenericType<T> { }
```

How to fix violations

To fix a violation of this rule, change the type to remove the generic type parameters or remove the `[GlobalClass]` attribute.

When to suppress warnings

Do not suppress a warning from this rule. Adding the `[GlobalClass]` to a generic type is an easy mistake to make and this warning helps users realize that it may result in unexpected errors.

C# API differences to GDScript

This is a (incomplete) list of API differences between C# and GDScript.

General differences

As explained in General differences between C# and GDScript, `PascalCase` is used to access Godot APIs in C# instead of the `snake_case` used by GDScript and C++. Where possible, fields and getters/setters have been converted to properties. In general, the C# Godot API strives to be as idiomatic as is reasonably possible. See the C# style guide, which we encourage you to also use for your own C# code.

In GDScript, the setters/getters of a property can be called directly, although this is not encouraged. In C#, only the property is defined. For example, to translate the GDScript code `x.set_name("Friend")` to C#, write `x.Name = "Friend";`.

A C# IDE will provide intellisense, which is extremely useful when figuring out renamed C# APIs. The built-in Godot script editor has no support for C# intellisense, and it also doesn't provide many other C# development tools that are considered essential. See Configuring an external editor.

Global scope

Global functions and some constants had to be moved to classes, since C# does not allow declaring them in namespaces. Most global constants were moved to their own enums.

Constants

In C#, only primitive types can be constant. For example, the `TAU` constant is replaced by the `Mathf.Tau` constant, but the `Vector2.RIGHT` constant is replaced by the `Vector2.Right` read-only property. This behaves similarly to a constant, but can't be used in some contexts like `switch` statements.

Global enum constants were moved to their own enums. For example, `ERR_*` constants were moved to the `Error` enum.

Special cases:

GDScript	C#
`TYPE_*`	`Variant.Type` enum
`OP_*`	`Variant.Operator` enum

Math functions

Math global functions, like `abs`, `acos`, `asin`, `atan` and `atan2`, are located under `Mathf` as `Abs`, `Acos`, `Asin`, `Atan` and `Atan2`. The `PI` constant can be found as `Mathf.Pi`.

C# also provides static System.Math [https://learn.microsoft.com/en-us/dotnet/api/system.math] and System.MathF [https://learn.microsoft.com/en-us/dotnet/api/system.mathf] classes that may contain other useful mathematical operations.

Random functions

Random global functions, like `rand_range` and `rand_seed`, are located under `GD`. Example: `GD.RandRange` and `GD.RandSeed`.

Consider using System.Random [https://learn.microsoft.com/en-us/dotnet/api/system.random] or, if you need cryptographically strong randomness, System.Security.Cryptography.RandomNumberGenerator [https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.randomnumbergenerator].

Other functions

Many other global functions like `print` and `var_to_str` are located under `GD`. Example: `GD.Print` and `GD.VarToStr`.

Exceptions:

GDScript	C#
`weakref(obj)`	`GodotObject.WeakRef(obj)`
`instance_from_id(id)`	`GodotObject.InstanceFromId(id)`
`is_instance_id_valid(id)`	`GodotObject.IsInstanceIdValid(id)`
`is_instance_valid(obj)`	`GodotObject.IsInstanceValid(obj)`

Tips

Sometimes it can be useful to use the `using static` directive. This directive allows to access the members and nested types of a class without specifying the class name.

Example:

```
using static Godot.GD;

public class Test
{
    static Test()
    {
        Print("Hello"); // Instead of GD.Print("Hello");
    }
}
```

Full list of equivalences

List of Godot's global scope functions and their equivalent in C#:

GDScript	C#
abs	Mathf.Abs
absf	Mathf.Abs
absi	Mathf.Abs
acos	Mathf.Acos
acosh	Mathf.Acosh
angle_difference	Mathf.AngleDifference
asin	Mathf.Asin

GDScript	C#
asinh	Mathf.Asinh
atan	Mathf.Atan
atan2	Mathf.Atan2
atanh	Mathf.Atanh
bezier_derivative	Mathf.BezierDerivative
bezier_interpolate	Mathf.BezierInterpolate
bytes_to_var	GD.BytesToVar
bytes_to_var_with_objects	GD.BytesToVarWithObjects
ceil	Mathf.Ceil
ceilf	Mathf.Ceil
ceili	Mathf.CeilToInt
clamp	Mathf.Clamp

GDScript	C#
clampf	Mathf.Clamp
clampi	Mathf.Clamp
cos	Mathf.Cos
cosh	Mathf.Cosh
cubic_interpolate	Mathf.CubicInterpolate
cubic_interpolate_angle	Mathf.CubicInterpolateAngle
cubic_interpolate_angle_in_time	Mathf.CubicInterpolateInTime
cubic_interpolate_in_time	Mathf.CubicInterpolateAngleInTime
db_to_linear	Mathf.DbToLinear
deg_to_rad	Mathf.DegToRad
ease	Mathf.Ease
error_string	Error.ToString

GDScript	C#
exp	Mathf.Exp
floor	Mathf.Floor
floorf	Mathf.Floor
floori	Mathf.FloorToInt
fmod	operator %
fposmod	Mathf.PosMod
hash	GD.Hash
instance_from_id	GodotObject.InstanceFromId
inverse_lerp	Mathf.InverseLerp
is_equal_approx	Mathf.IsEqualApprox
is_finite	Mathf.IsFinite or float.IsFinite or double.IsFinite
is_inf	Mathf.IsInf or float.IsInfinity or double.IsInfinity

GDScript	C#
is_instance_id_valid	GodotObject.IsInstanceIdValid
is_instance_valid	GodotObject.IsInstanceValid
is_nan	Mathf.IsNaN or float.IsNaN or double.IsNaN
is_same	operator == or object.ReferenceEquals
is_zero_approx	Mathf.IsZeroApprox
lerp	Mathf.Lerp
lerp_angle	Mathf.LerpAngle
lerpf	Mathf.Lerp
linear_to_db	Mathf.LinearToDb
log	Mathf.Log
max	Mathf.Max
maxf	Mathf.Max

GDScript	C#
maxi	Mathf.Max
min	Mathf.Min
minf	Mathf.Min
mini	Mathf.Min
move_toward	Mathf.MoveToward
nearest_po2	Mathf.NearestPo2
pingpong	Mathf.PingPong
posmod	Mathf.PosMod
pow	Mathf.Pow
print	GD.Print
print_rich	GD.PrintRich
print_verbose	Use OS.IsStdoutVerbose and GD.Print

GDScript	C#
printerr	GD.PrintErr
printraw	GD.PrintRaw
prints	GD.PrintS
printt	GD.PrintT
push_error	GD.PushError
push_warning	GD.PushWarning
rad_to_deg	Mathf.RadToDeg
rand_from_seed	GD.RandFromSeed
randf	GD.Randf
randf_range	GD.RandRange
randfn	GD.Randfn
randi	GD.Randi

GDScript	C#
randi_range	GD.RandRange
randomize	GD.Randomize
remap	Mathf.Remap
rid_allocate_id	N/A
rid_from_int64	N/A
rotate_toward	Mathf.RotateToward
round	Mathf.Round
roundf	Mathf.Round
roundi	Mathf.RoundToInt
seed	GD.Seed
sign	Mathf.Sign
signf	Mathf.Sign

GDScript	C#
signi	Mathf.Sign
sin	Mathf.Sin
sinh	Mathf.Sinh
smoothstep	Mathf.SmoothStep
snapped	Mathf.Snapped
snappedf	Mathf.Snapped
snappedi	Mathf.Snapped
sqrt	Mathf.Sqrt
step_decimals	Mathf.StepDecimals
str	Use $ string interpolation
str_to_var	GD.StrToVar
tan	Mathf.Tan

GDScript	C#
tanh	Mathf.Tanh
type_convert	Variant.As<T> or GD.Convert
type_string	Variant.Type.ToString
typeof	Variant.VariantType
var_to_bytes	GD.VarToBytes
var_to_bytes_with_objects	GD.VarToBytesWithObjects
var_to_str	GD.VarToStr
weakref	GodotObject.WeakRef
wrap	Mathf.Wrap
wrapf	Mathf.Wrap
wrapi	Mathf.Wrap

List of GDScript utility functions and their equivalent in C#:

GDScript	C#
assert	System.Diagnostics.Debug.Assert
char	Use explicit conversion: `(char)65`
convert	GD.Convert
dict_to_inst	N/A
get_stack	System.Environment.StackTrace
inst_to_dict	N/A
len	N/A
load	GD.Load
preload	N/A
print_debug	N/A
print_stack	GD.Print(System.Environment.StackTrace)
range	GD.Range or System.Linq.Enumerable.Range

GDScript	C#
type_exists	ClassDB.ClassExists(type)

`preload`, as it works in GDScript, is not available in C#. Use `GD.Load` or `ResourceLoader.Load` instead.

`@export` annotation

Use the `[Export]` attribute instead of the GDScript `@export` annotation. This attribute can also be provided with optional PropertyHint and `hintString` parameters. Default values can be set by assigning a value.

Example:

```
using Godot;

public partial class MyNode : Node
{
    [Export]
    private NodePath _nodePath;

    [Export]
    private string _name = "default";

    [Export(PropertyHint.Range, "0,100000,1000,or_greater")]
    private int _income;

    [Export(PropertyHint.File, "*.png,*.jpg")]
    private string _icon;
}
```

See also: C# exported properties.

`signal` keyword

Use the `[Signal]` attribute to declare a signal instead of the GDScript `signal` keyword. This attribute should be used on a *delegate*, whose name signature will be used to define the signal. The *delegate* must have the `EventHandler` suffix, an *event* will be generated in the class with the same name but without the suffix, use that event's name with `EmitSignal`.

```
[Signal]
delegate void MySignalEventHandler(string willSendAString);
```

See also: C# signals.

@onready annotation

GDScript has the ability to defer the initialization of a member variable until the ready function is called with *@onready* (cf. @onready annotation). For example:

```
@onready var my_label = get_node("MyLabel")
```

However C# does not have this ability. To achieve the same effect you need to do this.

```
private Label _myLabel;

public override void _Ready()
{
    _myLabel = GetNode<Label>("MyLabel");
}
```

Singletons

Singletons are available as static classes rather than using the singleton pattern. This is to make code less verbose than it would be with an `Instance` property.

Example:

```
Input.IsActionPressed("ui_down")
```

However, in some very rare cases this is not enough. For example, you may want to access a member from the base class `GodotObject`, like `Connect`. For such use cases we provide a static property named `Singleton` that returns the singleton instance. The type of this instance is `GodotObject`.

Example:

```
Input.Singleton.JoyConnectionChanged += Input_JoyConnectionChanged;
```

String

Use `System.String` (`string`). Most of Godot's String methods have an equivalent in `System.String` or are provided by the `StringExtensions` class as extension methods.

Example:

```
string text = "Get up!";
string[] bigrams = text.Bigrams(); // ["Ge", "et", "t ", " u",
"up", "p!"]
```

Strings are immutable in .NET, so all methods that manipulate a string don't modify the original string and return a newly created string with the modifications applied. To avoid creating multiple string allocations consider using a StringBuilder [https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder].

List of Godot's String methods and their equivalent in C#:

GDScript	C#
begins_with	string.StartsWith

GDScript	C#
bigrams	StringExtensions.Bigrams
bin_to_int	StringExtensions.BinToInt
c_escape	StringExtensions.CEscape
c_unescape	StringExtensions.CUnescape
capitalize	StringExtensions.Capitalize
casecmp_to	StringExtensions.CasecmpTo or StringExtensions.CompareTo (Consider using string.Equals or string.Compare)
chr	N/A
contains	string.Contains
count	StringExtensions.Count (Consider using RegEx)
countn	StringExtensions.CountN (Consider using RegEx)
dedent	StringExtensions.Dedent
ends_with	string.EndsWith

GDScript	C#
erase	string.Remove (Consider using StringBuilder to manipulate strings)
find	StringExtensions.Find (Consider using string.IndexOf or string.IndexOfAny)
findn	StringExtensions.FindN (Consider using string.IndexOf or string.IndexOfAny)
format	Use $ string interpolation
get_base_dir	StringExtensions.GetBaseDir
get_basename	StringExtensions.GetBaseName
get_extension	StringExtensions.GetExtension
get_file	StringExtensions.GetFile
get_slice	N/A
get_slice_count	N/A
get_slicec	N/A

GDScript	C#
hash	StringExtensions.Hash (Consider using object.GetHashCode unless you need to guarantee the same behavior as in GDScript)
hex_decode	StringExtensions.HexDecode (Consider using System.Convert.FromHexString)
hex_to_int	StringExtensions.HexToInt (Consider using int.Parse or long.Parse with System.Globalization.NumberStyles.HexNumber)
humanize_size	N/A
indent	StringExtensions.Indent
insert	string.Insert (Consider using StringBuilder to manipulate strings)
is_absolute_path	StringExtensions.IsAbsolutePath
is_empty	string.IsNullOrEmpty or string.IsNullOrWhiteSpace
is_relative_path	StringExtensions.IsRelativePath
is_subsequence_of	StringExtensions.IsSubsequenceOf

GDScript	C#
is_subsequence_ofn	StringExtensions.IsSubsequenceOfN
is_valid_filename	StringExtensions.IsValidFileName
is_valid_float	StringExtensions.IsValidFloat (Consider using float.TryParse or double.TryParse)
is_valid_hex_number	StringExtensions.IsValidHexNumber
is_valid_html_color	StringExtensions.IsValidHtmlColor
is_valid_identifier	StringExtensions.IsValidIdentifier
is_valid_int	StringExtensions.IsValidInt (Consider using int.TryParse or long.TryParse)
is_valid_ip_address	StringExtensions.IsValidIPAddress
join	string.Join
json_escape	StringExtensions.JSONEscape
left	StringExtensions.Left (Consider using string.Substring or string.AsSpan)

GDScript	C#
length	string.Length
lpad	string.PadLeft
lstrip	string.TrimStart
match	StringExtensions.Match (Consider using RegEx)
matchn	StringExtensions.MatchN (Consider using RegEx)
md5_buffer	StringExtensions.Md5Buffer (Consider using System.Security.Cryptography.MD5.HashData)
md5_text	StringExtensions.Md5Text (Consider using System.Security.Cryptography.MD5.HashData with StringExtensions.HexEncode)
naturalnocasecmp_to	N/A (Consider using string.Equals or string.Compare)
nocasecmp_to	StringExtensions.NocasecmpTo or StringExtensions.CompareTo (Consider using string.Equals or string.Compare)
num	float.ToString or double.ToString

GDScript	C#
num_int64	int.ToString or long.ToString
num_scientific	float.ToString or double.ToString
num_uint64	uint.ToString or ulong.ToString
pad_decimals	StringExtensions.PadDecimals
pad_zeros	StringExtensions.PadZeros
path_join	StringExtensions.PathJoin
repeat	Use string constructor or a StringBuilder
replace	string.Replace or RegEx
replacen	StringExtensions.ReplaceN (Consider using string.Replace or RegEx)
reverse	N/A
rfind	StringExtensions.RFind (Consider using string.LastIndexOf or string.LastIndexOfAny)

GDScript	C#
rfindn	StringExtensions.RFindN (Consider using string.LastIndexOf or string.LastIndexOfAny)
right	StringExtensions.Right (Consider using string.Substring or string.AsSpan)
rpad	string.PadRight
rsplit	N/A
rstrip	string.TrimEnd
sha1_buffer	StringExtensions.Sha1Buffer (Consider using System.Security.Cryptography.SHA1.HashData)
sha1_text	StringExtensions.Sha1Text (Consider using System.Security.Cryptography.SHA1.HashData with StringExtensions.HexEncode)
sha256_buffer	StringExtensions.Sha256Buffer (Consider using System.Security.Cryptography.SHA256.HashData)
sha256_text	StringExtensions.Sha256Text (Consider using System.Security.Cryptography.SHA256.HashData with StringExtensions.HexEncode)
similarity	StringExtensions.Similarity

GDScript	C#
simplify_path	StringExtensions.SimplifyPath
split	StringExtensions.Split (Consider using string.Split)
split_floats	StringExtensions.SplitFloat
strip_edges	StringExtensions.StripEdges (Consider using string.Trim, string.TrimStart or string.TrimEnd)
strip_escapes	StringExtensions.StripEscapes
substr	StringExtensions.Substr (Consider using string.Substring or string.AsSpan)
to_ascii_buffer	StringExtensions.ToAsciiBuffer (Consider using System.Text.Encoding.ASCII.GetBytes)
to_camel_case	StringExtensions.ToCamelCase
to_float	StringExtensions.ToFloat (Consider using float.TryParse or double.TryParse)
to_int	StringExtensions.ToInt (Consider using int.TryParse or long.TryParse)

GDScript	C#
to_lower	string.ToLower
to_pascal_case	StringExtensions.ToPascalCase
to_snake_case	StringExtensions.ToSnakeCase
to_upper	string.ToUpper
to_utf16_buffer	StringExtensions.ToUtf16Buffer (Consider using System.Text.Encoding.UTF16.GetBytes)
to_utf32_buffer	StringExtensions.ToUtf32Buffer (Consider using System.Text.Encoding.UTF32.GetBytes)
to_utf8_buffer	StringExtensions.ToUtf8Buffer (Consider using System.Text.Encoding.UTF8.GetBytes)
to_wchar_buffer	StringExtensions.ToUtf16Buffer in Windows and StringExtensions.ToUtf32Buffer in other platforms
trim_prefix	StringExtensions.TrimPrefix
trim_suffix	StringExtensions.TrimSuffix
unicode_at	string[int] indexer

GDScript	C#
uri_decode	StringExtensions.URIDecode (Consider using System.Uri.UnescapeDataString)
uri_encode	StringExtensions.URIEncode (Consider using System.Uri.EscapeDataString)
validate_node_name	StringExtensions.ValidateNodeName
xml_escape	StringExtensions.XMLEscape
xml_unescape	StringExtensions.XMLUnescape

List of Godot's PackedByteArray methods that create a String and their C# equivalent:

GDScript	C#
get_string_from_ascii	StringExtensions.GetStringFromAscii (Consider using System.Text.Encoding.ASCII.GetString)
get_string_from_utf16	StringExtensions.GetStringFromUtf16 (Consider using System.Text.Encoding.UTF16.GetString)
get_string_from_utf32	StringExtensions.GetStringFromUtf32 (Consider using System.Text.Encoding.UTF32.GetString)

GDScript	C#
get_string_from_utf8	StringExtensions.GetStringFromUtf8 (Consider using System.Text.Encoding.UTF8.GetString)
hex_encode	StringExtensions.HexEncode (Consider using System.Convert.ToHexString)

Note

.NET provides path utility methods under the System.IO.Path [https://learn.microsoft.com/en-us/dotnet/api/system.io.path] class. They can only be used with native OS paths, not Godot paths (paths that start with `res://` or `user://`). See File paths in Godot projects.

NodePath

The following method was converted to a property with a different name:

GDScript	C#
`is_empty()`	`IsEmpty`

Signal

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_name()`	`Name`
`get_object()`	`Owner`

The `Signal` type implements the awaitable pattern which means it can be used with the `await` keyword. See await keyword.

Instead of using the `Signal` type, the recommended way to use Godot signals in C# is to use the generated C# events. See C# signals.

Callable

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_object()`	`Target`
`get_method()`	`Method`

Currently C# supports `Callable` if one of the following holds:

- `Callable` was created using the C# `Callable` type.
- `Callable` is a basic version of the engine's `Callable`. Custom `Callable`s are unsupported. A `Callable` is custom when any of the following holds:

  - `Callable` has bound information (`Callable`s created with `bind`/`unbind` are unsupported).
  - `Callable` was created from other languages through the GDExtension API.

Some methods such as `bind` and `unbind` are not implemented, use lambdas instead:

```
string name = "John Doe";
Callable callable = Callable.From(() => SayHello(name));

void SayHello(string name)
{
    GD.Print($"Hello {name}");
}
```

The lambda captures the `name` variable so it can be bound to the `SayHello` method.

RID

This type is named `Rid` in C# to follow the .NET naming convention.

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_id()`	`Id`
`is_valid()`	`IsValid`

Basis

Structs cannot have parameterless constructors in C#. Therefore, `new Basis()` initializes all primitive members to their default value. Use `Basis.Identity` for the equivalent of `Basis()` in GDScript and C++.

The following method was converted to a property with a different name:

GDScript	C#
`get_scale()`	`Scale`

Transform2D

Structs cannot have parameterless constructors in C#. Therefore, `new Transform2D()` initializes all primitive members to their default value. Please use `Transform2D.Identity` for the equivalent of `Transform2D()` in GDScript and C++.

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_rotation()`	`Rotation`
`get_scale()`	`Scale`
`get_skew()`	`Skew`

Transform3D

Structs cannot have parameterless constructors in C#. Therefore, `new Transform3D()` initializes all primitive members to their default value. Please use `Transform3D.Identity` for the equivalent of `Transform3D()` in GDScript and C++.

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_rotation()`	`Rotation`
`get_scale()`	`Scale`

Rect2

The following field was converted to a property with a *slightly* different name:

GDScript	C#
`end`	`End`

The following method was converted to a property with a different name:

GDScript	C#
`get_area()`	`Area`

Rect2i

This type is named `Rect2I` in C# to follow the .NET naming convention.

The following field was converted to a property with a *slightly* different name:

GDScript	C#
`end`	`End`

The following method was converted to a property with a different name:

GDScript	C#
`get_area()`	`Area`

AABB

This type is named `Aabb` in C# to follow the .NET naming convention.

The following method was converted to a property with a different name:

GDScript	C#
`get_volume()`	`Volume`

Quaternion

Structs cannot have parameterless constructors in C#. Therefore, `new Quaternion()` initializes all primitive members to their default value. Please use `Quaternion.Identity` for the equivalent of `Quaternion()` in GDScript and C++.

Projection

Structs cannot have parameterless constructors in C#. Therefore, `new Projection()` initializes all primitive members to their default value. Please use `Projection.Identity` for the equivalent of `Projection()` in GDScript and C++.

Color

Structs cannot have parameterless constructors in C#. Therefore, `new Color()` initializes all primitive members to their default value (which represents the transparent black color). Please use `Colors.Black` for the equivalent of `Color()` in GDScript and C++.

The global `Color8` method to construct a Color from bytes is available as a static method in the Color type.

The Color constants are available in the `Colors` static class as readonly properties.

The following method was converted to a property with a different name:

GDScript	C#
`get_luminance()`	`Luminance`

The following method was converted to a method with a different name:

GDScript	C#
`html(String)`	`FromHtml(ReadOnlySpan<char>)`

The following methods are available as constructors:

GDScript	C#
`hex(int)`	`Color(uint)`
`hex64(int)`	`Color(ulong)`

Array

The equivalent of packed arrays are `System.Array`.

See also PackedArray in C#.

Use `Godot.Collections.Array` for an untyped `Variant` array. `Godot.Collections.Array<T>` is a type-safe wrapper around

`Godot.Collections.Array`.

See also Array in C#.

Dictionary

Use `Godot.Collections.Dictionary` for an untyped `Variant` dictionary. `Godot.Collections.Dictionary<TKey, TValue>` is a type-safe wrapper around `Godot.Collections.Dictionary`.

See also Dictionary in C#.

Variant

`Godot.Variant` is used to represent Godot's native Variant type. Any Variant-compatible type can be converted from/to it.

See also: C# Variant.

Communicating with other scripting languages

This is explained extensively in Cross-language scripting.

`await` keyword

Something similar to GDScript's `await` keyword can be achieved with C#'s await keyword [https://docs.microsoft.com/en-US/dotnet/csharp/language-reference/keywords/await].

The `await` keyword in C# can be used with any awaitable expression. It's commonly used with operands of the types Task [https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task],

Task<TResult> [https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1], ValueTask [https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask], or ValueTask<TResult> [https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1].

An expression `t` is awaitable if one of the following holds:

- `t` is of compile-time type `dynamic`.
- `t` has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type `A` for which all of the following hold:
  - `A` implements the interface `System.Runtime.CompilerServices.INotifyCompletion`.
  - `A` has an accessible, readable instance property `IsCompleted` of type `bool`.
  - `A` has an accessible instance method `GetResult` with no parameters and no type parameters.

An equivalent of awaiting a signal in GDScript can be achieved with the `await` keyword and `GodotObject.ToSignal`.

Example:

```
public async Task SomeFunction()
{
    await ToSignal(timer, Timer.SignalName.Timeout);
    GD.Print("After timeout");
}
```

C# collections

The .NET base class library contains multiple collection types that can be used to store and manipulate data. Godot also provide some collection types that are tightly integrated with the rest of the engine.

Choose a collection

The main difference between the .NET collections [https://learn.microsoft.com/en-us/dotnet/standard/collections/] and the Godot collections is that the .NET collections are implemented in C# while the Godot collections are implemented in C++ and the Godot C# API is a wrapper over it, this is an important distinction since it means every operation on a Godot collection requires marshaling which can be expensive especially inside a loop.

Due to the performance implications, using Godot collections is only recommended when absolutely necessary (such as interacting with the Godot API). Godot only understands its own collection types, so it's required to use them when talking to the engine.

If you have a collection of elements that don't need to be passed to a Godot API, using a .NET collection would be more performant.

Tip

It's also possible to convert between .NET collections and Godot collections. The Godot collections contain constructors from generic .NET collection interfaces that copy their elements, and the Godot collections can be used with the LINQ [https://learn.microsoft.com/en-us/dotnet/standard/linq] `ToList`, `ToArray` and `ToDictionary` methods. But keep in mind this conversion requires marshaling every element in the collection and copies it to a new collection so it can be expensive.

Despite this, the Godot collections are optimized to try and avoid unnecessary marshaling, so methods like `Sort` or `Reverse` are implemented with a single interop call and don't need to marshal every element. Keep an eye out for generic APIs that take collection interfaces like LINQ [https://learn.microsoft.com/en-us/dotnet/standard/linq] because every method requires iterating the collection and, therefore, marshaling every element. Prefer using the instance methods of the Godot collections when possible.

To choose which collection type to use for each situation, consider the following questions:

- Does your collection need to interact with the Godot engine? (e.g.: the type of an exported property, calling a Godot method).
  - If yes, since Godot only supports Variant-compatible types, use a Godot collection.
  - If not, consider choosing an appropriate .NET collection [https://learn.microsoft.com/en-us/dotnet/standard/collections/selecting-a-collection-class].
- Do you need a Godot collection that represents a list or sequential set of data?
  - Godot arrays are similar to the C# collection `List<T>`.
  - Godot packed arrays are more memory-efficient arrays, in C# use one of the supported `System.Array` types.
- Do you need a Godot collection that maps a set of keys to a set of values?

    - Godot dictionaries store pairs of keys and values and allow easy access to the values by their associated key.

Godot collections

PackedArray

Godot packed arrays are implemented as an array of a specific type, allowing it to be more tightly packed as each element has the size of the specific type, not `Variant`.

In C#, packed arrays are replaced by `System.Array`:

GDScript	C#
`PackedByteArray`	`byte[]`
`PackedInt32Array`	`int[]`
`PackedInt64Array`	`long[]`
`PackedFloat32Array`	`float[]`
`PackedFloat64Array`	`double[]`
`PackedStringArray`	`string[]`
`PackedVector2Array`	`Vector2[]`

GDScript	C#
`PackedVector3Array`	`Vector3[]`
`PackedVector4Array`	`Vector4[]`
`PackedColorArray`	`Color[]`

Other C# arrays are not supported by the Godot C# API since a packed array equivalent does not exist. See the list of Variant-compatible types.

Array

Godot arrays are implemented as an array of `Variant` and can contain several elements of any type. In C#, the equivalent type is `Godot.Collections.Array`.

The generic `Godot.Collections.Array<T>` type allows restricting the element type to a Variant-compatible type.

An untyped `Godot.Collections.Array` can be converted to a typed array using the `Godot.Collections.Array<T>(Godot.Collections.Array)` constructor.

Note

Despite the name, Godot arrays are more similar to the C# collection `List<T>` than `System.Array`. Their size is not fixed and can grow or shrink as elements are added/removed from the collection.

List of Godot's Array methods and their equivalent in C#:

GDScript	C#
all	System.Linq.Enumerable.All
any	System.Linq.Enumerable.Any
append	Add
append_array	AddRange
assign	Clear and AddRange
back	`Array[^1]` or System.Linq.Enumerable.Last or System.Linq.Enumerable.LastOrDefault
bsearch	BinarySearch
bsearch_custom	N/A
clear	Clear

GDScript	C#
count	System.Linq.Enumerable.Count
duplicate	Duplicate
erase	Remove
fill	Fill
filter	Use System.Linq.Enumerable.Where
find	IndexOf
front	`Array[0]` or System.Linq.Enumerable.First or System.Linq.Enumerable.FirstOrDefault
get_typed_builtin	N/A
get_typed_class_name	N/A
get_typed_script	N/A
has	Contains

GDScript	C#
hash	GD.Hash
insert	Insert
is_empty	Use `Count == 0`
is_read_only	IsReadOnly
is_same_typed	N/A
is_typed	N/A
make_read_only	MakeReadOnly
map	System.Linq.Enumerable.Select
max	Max
min	Min
pick_random	PickRandom (Consider using System.Random)

GDScript	C#
pop_at	`Array[i]` with `RemoveAt(i)`
pop_back	`Array[^1]` with `RemoveAt(Count - 1)`
pop_front	`Array[0]` with `RemoveAt(0)`
push_back	`Insert(Count, item)`
push_front	`Insert(0, item)`
reduce	System.Linq.Enumerable.Aggregate
remove_at	RemoveAt
resize	Resize
reverse	Reverse
rfind	LastIndexOf
shuffle	Shuffle
size	Count

GDScript	C#
slice	Slice
sort	Sort
sort_custom	System.Linq.Enumerable.OrderBy
operator !=	!RecursiveEqual
operator +	operator +
operator <	N/A
operator <=	N/A
operator ==	RecursiveEqual
operator >	N/A
operator >=	N/A
operator []	Array[int] indexer

Dictionary

Godot dictionaries are implemented as a dictionary with `Variant` keys and values. In C#, the equivalent type is `Godot.Collections.Dictionary`.

The generic `Godot.Collections.Dictionary<TKey, TValue>` type allows restricting the key and value types to a Variant-compatible type.

An untyped `Godot.Collections.Dictionary` can be converted to a typed dictionary using the `Godot.Collections.Dictionary<TKey, TValue>(Godot.Collections.Dictionary)` constructor.

Tip

If you need a dictionary where the key is typed but not the value, use `Variant` as the `TValue` generic parameter of the typed dictionary.

```
// The keys must be string, but the values can be any Variant-compatible type.
var dictionary = new Godot.Collections.Dictionary<string, Variant>();
```

List of Godot's Dictionary methods and their equivalent in C#:

GDScript	C#
clear	Clear
duplicate	Duplicate

GDScript	C#
erase	Remove
find_key	N/A
get	Dictionary[Variant] indexer or TryGetValue
has	ContainsKey
has_all	N/A
hash	GD.Hash
is_empty	Use `Count == 0`
is_read_only	IsReadOnly
keys	Keys
make_read_only	MakeReadOnly
merge	Merge
size	Count

GDScript	C#
values	Values
operator !=	!RecursiveEqual
operator ==	RecursiveEqual
operator []	Dictionary[Variant] indexer, Add or TryGetValue

C# Variant

For a detailed explanation of Variant in general, see the Variant documentation page.

`Godot.Variant` is used to represent Godot's native Variant type. Any Variant-compatible type can be converted from/to it. We recommend avoiding `Godot.Variant` unless it is necessary to interact with untyped engine APIs. Take advantage of C#'s type safety when possible.

Converting from a Variant-compatible C# type to `Godot.Variant` can be done using implicit conversions. There are also `CreateFrom` method overloads and the generic `Variant.From<T>` methods. Only the syntax is different: the behavior is the same.

```
int x = 42;
Variant numberVariant = x;
Variant helloVariant = "Hello, World!";

Variant numberVariant2 = Variant.CreateFrom(x);
Variant numberVariant3 = Variant.From(x);
```

Implicit conversions to `Godot.Variant` make passing variants as method arguments very convenient. For example, the third argument of tween_property specifying the final color of the tween is a `Godot.Variant`.

```
Tween tween = CreateTween();
tween.TweenProperty(GetNode("Sprite"), "modulate", Colors.Red,
1.0f);
```

Converting from `Godot.Variant` to a C# type can be done using explicit conversions. There are also `Variant.As{TYPE}` methods and the generic `Variant.As<T>` method. All of these behave the same.

```
int number = (int)numberVariant;
string hello = (string)helloVariant;

int number2 = numberVariant.As<int>();
int number3 = numberVariant.AsInt32();
```

Note

The `Variant.As{TYPE}` methods are typically named after C# types (`Int32`), not C# keywords (`int`).

If the Variant type doesn't match the conversion target type, the consequences vary depending on the source and target values.

- The conversion may examine the value and return a similar but potentially unexpected value of the target type. For example, the string `"42a"` may be converted to the integer `42`.
- The default value of the target type may be returned.
- An empty array may be returned.
- An exception may be thrown.

Converting to the correct type avoids complicated behavior and should be preferred.

The `Variant.Obj` property returns a C# `object` with the correct value for any variant. This may be useful when the type of Variant is completely unknown. However, when possible, prefer more specific conversions. `Variant.Obj` evaluates a `switch` on `Variant.VariantType` and it may not be necessary. Also, if the result is a value type, it is boxed.

For example, if the potential for `Variant.As<MyNode>()` to throw an invalid cast exception isn't acceptable, consider using a `Variant.As<GodotObject>() is MyNode n` type pattern instead.

Note

Since the Variant type in C# is a struct, it can't be null. To create a "null" Variant, use the `default` keyword or the `Godot.Variant` parameterless constructor.

Variant-compatible types

A Variant-compatible type can be converted to and from a `Godot.Variant`. These C# types are Variant-compatible:

- All the built-in value types [https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/built-in-types-table], except `decimal`, `nint` and `nuint`.
- `string`.
- Classes derived from GodotObject.
- Collections types defined in the `Godot.Collections` namespace.

Full list of Variant types and their equivalent C# type:

Variant.Type	C# Type
`Nil`	`null` (Not a type)
`Bool`	`bool`
`Int`	`long` (Godot stores 64-bit integers in Variant)
`Float`	`double` (Godot stores 64-bit floats in Variant)
`String`	`string`

Variant.Type	C# Type
`Vector2`	`Godot.Vector2`
`Vector2I`	`Godot.Vector2I`
`Rect2`	`Godot.Rect2`
`Rect2I`	`Godot.Rect2I`
`Vector3`	`Godot.Vector3`
`Vector3I`	`Godot.Vector3I`
`Transform2D`	`Godot.Transform2D`
`Vector4`	`Godot.Vector4`
`Vector4I`	`Godot.Vector4I`
`Plane`	`Godot.Plane`
`Quaternion`	`Godot.Quaternion`

Variant.Type	**C# Type**
`Aabb`	`Godot.Aabb`
`Basis`	`Godot.Basis`
`Transform3D`	`Godot.Transform3D`
`Projection`	`Godot.Projection`
`Color`	`Godot.Color`
`StringName`	`Godot.StringName`
`NodePath`	`Godot.NodePath`
`Rid`	`Godot.Rid`
`Object`	`Godot.GodotObject` or any derived type.
`Callable`	`Godot.Callable`
`Signal`	`Godot.Signal`
`Dictionary`	`Godot.Collections.Dictionary`

Variant.Type	C# Type
`Array`	`Godot.Collections.Array`
`PackedByteArray`	`byte[]`
`PackedInt32Array`	`int[]`
`PackedInt64Array`	`long[]`
`PackedFloat32Array`	`float[]`
`PackedFloat64Array`	`double[]`
`PackedStringArray`	`string[]`
`PackedVector2Array`	`Godot.Vector2[]`
`PackedVector3Array`	`Godot.Vector3[]`
`PackedVector4Array`	`Godot.Vector4[]`
`PackedColorArray`	`Godot.Color[]`

Warning

Godot uses 64-bit integers and floats in Variant. Smaller integer and float types such as `int`, `short` and `float` are supported since they can fit in the bigger type. Be aware that when a conversion is performed, using the wrong type will result in potential precision loss.

Warning

Enums are supported by `Godot.Variant` since their underlying type is an integer type which are all compatible. However, implicit conversions don't exist, enums must be manually converted to their underlying integer type before they can converted to/from `Godot.Variant` or use the generic `Variant.As<T>` and `Variant.From<T>` methods to convert them.

```
enum MyEnum { A, B, C }

Variant variant1 = (int)MyEnum.A;
MyEnum enum1 = (MyEnum)(int)variant1;

Variant variant2 = Variant.From(MyEnum.A);
MyEnum enum2 = variant2.As<MyEnum>();
```

Using Variant in a generic context

When using generics, you may be interested in restricting the generic `T` type to be only one of the Variant-compatible types. This can be achieved using the `[MustBeVariant]` attribute.

```
public void MethodThatOnlySupportsVariants<[MustBeVariant] T>(T
onlyVariant)
{
    // Do something with the Variant-compatible value.
}
```

Combined with the generic `Variant.From<T>` allows you to obtain an instance of `Godot.Variant` from an instance of a generic `T` type. Then it can be used in any API that only supports the `Godot.Variant` struct.

```
public void Method1<[MustBeVariant] T>(T variantCompatible)
{
    Variant variant = Variant.From(variantCompatible);
    Method2(variant);
}

public void Method2(Variant variant)
{
    // Do something with variant.
}
```

In order to invoke a method with a generic parameter annotated with the `[MustBeVariant]` attribute, the value must be a Variant-compatible type or a generic `T` type annotated with the `[MustBeVariant]` attribute as well.

```
public class ObjectDerivedClass : GodotObject { }

public class NonObjectDerivedClass { }

public void Main<[MustBeVariant] T1, T2>(T1 someGeneric1, T2 someGeneric2)
{
    MyMethod(42); // Works because `int` is a Variant-compatible type.
    MyMethod(new ObjectDerivedClass()); // Works because any type that derives from `GodotObject` is a Variant-compatible type.
    MyMethod(new NonObjectDerivedClass()); // Does NOT work because the type is not Variant-compatible.
    MyMethod(someGeneric1); // Works because `T1` is annotated with the `[MustBeVariant]` attribute.
    MyMethod(someGeneric2); // Does NOT work because `T2` is
NOT annotated with the `[MustBeVariant]` attribute.
}

public void MyMethod<[MustBeVariant] T>(T variant)
{
    // Do something with variant.
}
```

C# signals

For a detailed explanation of signals in general, see the Using signals section in the step by step tutorial.

Signals are implemented using C# events, the idiomatic way to represent the observer pattern in C#. This is the recommended way to use signals in C# and the focus of this page.

In some cases it's necessary to use the older Connect() and Disconnect() APIs. See Using Connect and Disconnect for more details.

If you encounter a `System.ObjectDisposedException` while handling a signal, you might be missing a signal disconnection. See Disconnecting automatically when the receiver is freed for more details.

Signals as C# events

To provide more type-safety, Godot signals are also all available through events [https://learn.microsoft.com/en-us/dotnet/csharp/events-overview]. You can handle these events, as any other event, with the `+=` and `-=` operators.

```
Timer myTimer = GetNode<Timer>("Timer");
myTimer.Timeout += () => GD.Print("Timeout!");
```

In addition, you can always access signal names associated with a node type through its nested `SignalName` class. This is useful when, for example, you want to await on a signal (see await keyword).

```
await ToSignal(GetTree(), SceneTree.SignalName.ProcessFrame);
```

Custom signals as C# events

To declare a custom event in your C# script, use the `[Signal]` attribute on a public delegate type. Note that the name of this delegate needs to end with `EventHandler`.

```
[Signal]
public delegate void MySignalEventHandler();

[Signal]
public delegate void MySignalWithArgumentEventHandler(string
myString);
```

Once this is done, Godot will create the appropriate events automatically behind the scenes. You can then use said events as you'd do for any other Godot signal. Note that events are named using your delegate's name minus the final `EventHandler` part.

```
public override void _Ready()
{
    MySignal += () => GD.Print("Hello!");
    MySignalWithArgument += SayHelloTo;
}

private void SayHelloTo(string name)
{
    GD.Print($"Hello {name}!");
}
```

Warning

If you want to connect to these signals in the editor, you will need to (re)build the project to see them appear.

You can click the **Build** button in the upper-right corner of the editor to do so.

Signal emission

To emit signals, use the `EmitSignal` method. Note that, as for signals defined by the engine, your custom signal names are listed under the nested `SignalName` class.

```
public void MyMethodEmittingSignals()
{
    EmitSignal(SignalName.MySignal);
    EmitSignal(SignalName.MySignalWithArgument, "World");
}
```

In contrast with other C# events, you cannot use `Invoke` to raise events tied to Godot signals.

Signals support arguments of any Variant-compatible type.

Consequently, any `Node` or `RefCounted` will be compatible automatically, but custom data objects will need to inherit from `GodotObject` or one of its subclasses.

```
using Godot;

public partial class DataObject : GodotObject
{
    public string MyFirstString { get; set; }
    public string MySecondString { get; set; }
}
```

Bound values

Sometimes you'll want to bind values to a signal when the connection is established, rather than (or in addition to) when the signal is emitted. To do so, you can use an anonymous function like in the following example.

Here, the Button.Pressed signal does not take any argument. But we want to use the same `ModifyValue` for both the "plus" and "minus" buttons. So we bind the modifier value at the time we're connecting the signals.

```
public int Value { get; private set; } = 1;

public override void _Ready()
{
    Button plusButton = GetNode<Button>("PlusButton");
    plusButton.Pressed += () => ModifyValue(1);

    Button minusButton = GetNode<Button>("MinusButton");
    minusButton.Pressed += () => ModifyValue(-1);
}

private void ModifyValue(int modifier)
{
    Value += modifier;
}
```

Signal creation at runtime

Finally, you can create custom signals directly while your game is running. Use the `AddUserSignal` method for that. Be aware that it should be executed before any use of said signals (either connecting to them or emitting them). Also, note that signals created this way won't be visible through the `SignalName` nested class.

```
public override void _Ready()
{
    AddUserSignal("MyCustomSignal");
    EmitSignal("MyCustomSignal");
}
```

Using Connect and Disconnect

In general, it isn't recommended to use Connect() and Disconnect(). These APIs don't provide as much type safety as the events. However, they're necessary for connecting to signals defined by GDScript and passing ConnectFlags.

In the following example, pressing the button for the first time prints `Greetings!`. `OneShot` disconnects the signal, so pressing the button again does nothing.

```
public override void _Ready()
{
    Button button = GetNode<Button>("GreetButton");
    button.Connect(Button.SignalName.Pressed,
Callable.From(OnButtonPressed),
(uint)GodotObject.ConnectFlags.OneShot);
}

public void OnButtonPressed()
{
    GD.Print("Greetings!");
}
```

Disconnecting automatically when the receiver is freed

Normally, when any `GodotObject` is freed (such as any `Node`), Godot automatically disconnects all connections associated with that object. This happens for both signal emitters and signal receivers.

For example, a node with this code will print "Hello!" when the button is pressed, then free itself. Freeing the node disconnects the signal, so pressing the button again doesn't do anything:

```
public override void _Ready()
{
    Button myButton = GetNode<Button>("../MyButton");
    myButton.Pressed += SayHello;
}

private void SayHello()
{
    GD.Print("Hello!");
    Free();
}
```

When a signal receiver is freed while the signal emitter is still alive, in some cases automatic disconnection won't happen:

- The signal is connected to a lambda expression that captures a variable.
- The signal is a custom signal.

The following sections explain these cases in more detail and include suggestions for how to disconnect manually.

Note

Automatic disconnection is totally reliable if a signal emitter is freed before any of its receivers are freed. With a project style that prefers this pattern, the above limits may not be a concern.

No automatic disconnection: a lambda expression that captures a variable

If you connect to a lambda expression that captures variables, Godot can't tell that the lambda is associated with the instance that created it. This causes this example to have potentially unexpected behavior:

```
Timer myTimer = GetNode<Timer>("../Timer");
int x = 0;
myTimer.Timeout += () =>
{
    x++; // This lambda expression captures x.
    GD.Print($"Tick {x} my name is {Name}");
    if (x == 3)
    {
        GD.Print("Time's up!");
        Free();
    }
};
```

```
Tick 1, my name is ExampleNode
Tick 2, my name is ExampleNode
Tick 3, my name is ExampleNode
Time's up!
[...] System.ObjectDisposedException: Cannot access a disposed
object.
```

On tick 4, the lambda expression tries to access the `Name` property of the node, but the node has already been freed. This causes the exception.

To disconnect, keep a reference to the delegate created by the lambda expression and pass that to `-=`. For example, this node connects and disconnects using the `_EnterTree` and `_ExitTree` lifecycle methods:

```
[Export]
public Timer MyTimer { get; set; }

private Action _tick;

public override void _EnterTree()
{
    int x = 0;
    _tick = () =>
    {
        x++;
        GD.Print($"Tick {x} my name is {Name}");
        if (x == 3)
        {
            GD.Print("Time's up!");
            Free();
        }
    };
    MyTimer.Timeout += _tick;
}

public override void _ExitTree()
{
    MyTimer.Timeout -= _tick;
}
```

In this example, `Free` causes the node to leave the tree, which calls `_ExitTree`. `_ExitTree` disconnects the signal, so `_tick` is never called again.

The lifecycle methods to use depend on what the node does. Another option is to connect to signals in `_Ready` and disconnect in `Dispose`.

Note

Godot uses Delegate.Target [https://learn.microsoft.com/en-us/dotnet/api/system.delegate.target] to determine what instance a delegate is associated with. When a lambda expression doesn't capture a variable, the generated delegate's `Target` is the instance that created the delegate. When a variable is captured, the `Target` instead points at a generated type that stores the captured variable. This is what breaks the association. If you want to see if a delegate will be automatically cleaned up, try checking its `Target`.

`Callable.From` doesn't affect the `Delegate.Target`, so connecting a lambda that captures variables using `Connect` doesn't work any better than `+=`.

No automatic disconnection: a custom signal

Connecting to a custom signal using `+=` doesn't disconnect automatically when the receiving node is freed.

To disconnect, use `-=` at an appropriate time. For example:

```
[Export]
public MyClass Target { get; set; }

public override void _EnterTree()
{
    Target.MySignal += OnMySignal;
}

public override void _ExitTree()
{
    Target.MySignal -= OnMySignal;
}
```

Another solution is to use `Connect`, which does disconnect automatically with custom signals:

```
[Export]
public MyClass Target { get; set; }

public override void _EnterTree()
{
    Target.Connect(MyClass.SignalName.MySignal,
Callable.From(OnMySignal));
}
```

C# exported properties

In Godot, class members can be exported. This means their value gets saved along with the resource (such as the scene) they're attached to. They will also be available for editing in the property editor. Exporting is done by using the `[Export]` attribute.

```
using Godot;

public partial class ExportExample : Node3D
{
    [Export]
    public int Number { get; set; } = 5;
}
```

In that example the value `5` will be saved, and after building the current project it will be visible in the property editor.

One of the fundamental benefits of exporting member variables is to have them visible and editable in the editor. This way, artists and game designers can modify values that later influence how the program runs. For this, a special export syntax is provided.

Exporting can only be done with Variant-compatible types.

Note

Exporting properties can also be done in GDScript, for information on that see GDScript exported properties.

Basic use

Exporting works with fields and properties. They can have any access modifier.

```
[Export]
private int _number;

[Export]
public int Number { get; set; }
```

Exported members can specify a default value; otherwise, the default value of the type [https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/default-values] is used instead.

An `int` like `Number` defaults to `0`. `Text` defaults to null because `string` is a reference type.

```
[Export]
public int Number { get; set; }

[Export]
public string Text { get; set; }
```

Default values can be specified for fields and properties.

```
[Export]
private string _greeting = "Hello World";

[Export]
public string Greeting { get; set; } = "Hello World";
```

Properties with a backing field use the default value of the backing field.

```
private int _number = 2;

[Export]
public int NumberWithBackingField
{
    get => _number;
    set => _number = value;
}
```

Note

A property's `get` is not actually executed to determine the default value. Instead, Godot analyzes the C# source code. This works fine for most cases, such as the examples on this page. However, some properties are too complex for the analyzer to understand.

For example, the following property attempts to use math to display the default value as `5` in the property editor, but it doesn't work:

```
[Export]
public int NumberWithBackingField
{
    get => _number + 3;
    set => _number = value - 3;
}

private int _number = 2;
```

The analyzer doesn't understand this code and falls back to the default value for `int`, `0`. However, when running the scene or inspecting a node with an attached tool script, `_number` will be `2`, and `NumberWithBackingField` will return `5`. This difference may cause confusing behavior. To avoid this, don't use complex properties. Alternatively, if the default value can be explicitly specified, it can be overridden with the _PropertyCanRevert() and _PropertyGetRevert() methods.

Any type of `Resource` or `Node` can be exported. The property editor shows a user-friendly assignment dialog for these types. This can be used instead of `GD.Load` and `GetNode`. See Nodes and Resources.

```
[Export]
public PackedScene PackedScene { get; set; }

[Export]
public RigidBody2D RigidBody2D { get; set; }
```

Grouping exports

It is possible to group your exported properties inside the Inspector with the `[ExportGroup]` attribute. Every exported property after this attribute will be added to the group. Start a new group or use `[ExportGroup("")]` to break out.

```
[ExportGroup("My Properties")]
[Export]
public int Number { get; set; } = 3;
```

The second argument of the attribute can be used to only group properties with the specified prefix.

Groups cannot be nested, use `[ExportSubgroup]` to create subgroups within a group.

```
[ExportSubgroup("Extra Properties")]
[Export]
public string Text { get; set; } = "";
[Export]
public bool Flag { get; set; } = false;
```

You can also change the name of your main category, or create additional categories in the property list with the `[ExportCategory]` attribute.

```
[ExportCategory("Main Category")]
[Export]
public int Number { get; set; } = 3;
[Export]
public string Text { get; set; } = "";
```

```
[ExportCategory("Extra Category")]
[Export]
public bool Flag { get; set; } = false;
```

Note

The list of properties is organized based on the class inheritance, and new categories break that expectation. Use them carefully, especially when creating projects for public use.

Strings as paths

Property hints can be used to export strings as paths

String as a path to a file.

```
[Export(PropertyHint.File)]
public string GameFile { get; set; }
```

String as a path to a directory.

```
[Export(PropertyHint.Dir)]
public string GameDirectory { get; set; }
```

String as a path to a file, custom filter provided as hint.

```
[Export(PropertyHint.File, "*.txt,")]
public string GameFile { get; set; }
```

Using paths in the global filesystem is also possible, but only in scripts in tool mode.

String as a path to a PNG file in the global filesystem.

```
[Export(PropertyHint.GlobalFile, "*.png")]
public string ToolImage { get; set; }
```

String as a path to a directory in the global filesystem.

```
[Export(PropertyHint.GlobalDir)]
public string ToolDir { get; set; }
```

The multiline annotation tells the editor to show a large input field for editing over multiple lines.

```
[Export(PropertyHint.MultilineText)]
public string Text { get; set; }
```

Limiting editor input ranges

Using the range property hint allows you to limit what can be input as a value using the editor.

Allow integer values from 0 to 20.

```
[Export(PropertyHint.Range, "0,20,")]
public int Number { get; set; }
```

Allow integer values from -10 to 20.

```
[Export(PropertyHint.Range, "-10,20,")]
public int Number { get; set; }
```

Allow floats from -10 to 20 and snap the value to multiples of 0.2.

```
[Export(PropertyHint.Range, "-10,20,0.2")]
public float Number { get; set; }
```

If you add the hints "or_greater" and/or "or_less" you can go above or below the limits when adjusting the value by typing it instead of using the slider.

```
[Export(PropertyHint.Range, "0,100,1,or_greater,or_less")]
public int Number { get; set; }
```

Floats with easing hint

Display a visual representation of the 'ease()' function when editing.

```
[Export(PropertyHint.ExpEasing)]
public float TransitionSpeed { get; set; }
```

Colors

Regular color given as red-green-blue-alpha value.

```
[Export]
public Color Color { get; set; }
```

Color given as red-green-blue value (alpha will always be 1).

```
[Export(PropertyHint.ColorNoAlpha)]
public Color Color { get; set; }
```

Nodes

Since Godot 4.0, nodes can be directly exported without having to use NodePaths.

```
[Export]
public Node Node { get; set; }
```

A specific type of node can also be directly exported. The list of nodes shown after pressing "Assign" in the inspector is filtered to the specified type, and only a correct node can be assigned.

```
[Export]
public Sprite2D Sprite2D { get; set; }
```

Custom node classes can also be exported directly. The filtering behavior depends on whether the custom class is a global class.

Exporting NodePaths like in Godot 3.x is still possible, in case you need it:

```
[Export]
public NodePath NodePath { get; set; }

public override void _Ready()
{
    var node = GetNode(NodePath);
}
```

Resources

```
[Export]
public Resource Resource { get; set; }
```

In the Inspector, you can then drag and drop a resource file from the FileSystem dock into the variable slot.

Opening the inspector dropdown may result in an extremely long list of possible classes to create, however. Therefore, if you specify a type derived from Resource such as:

```
[Export]
public AnimationNode AnimationNode { get; set; }
```

The drop-down menu will be limited to AnimationNode and all its inherited classes. Custom resource classes can also be used, see C# global classes.

It must be noted that even if the script is not being run while in the editor, the exported properties are still editable. This can be used in conjunction with a script in "tool" mode.

Exporting bit flags

Members whose type is an enum with the `[Flags]` attribute can be exported and their values are limited to the members of the enum type. The editor will create a widget in the Inspector, allowing to select none, one, or multiple of the enum members. The value will be stored as an integer.

A flags enum uses powers of 2 for the values of the enum members. Members that combine multiple flags using logical OR (`|`) are also possible.

```
[Flags]
public enum MyEnum
{
    Fire = 1 << 1,
    Water = 1 << 2,
    Earth = 1 << 3,
    Wind = 1 << 4,

    FireAndWater = Fire | Water,
}

[Export]
public SpellElements MySpellElements { get; set; }
```

Integers used as bit flags can store multiple `true`/`false` (boolean) values in one property. By using the `Flags` property hint, any of the given flags can be set from the editor.

```
[Export(PropertyHint.Flags, "Fire,Water,Earth,Wind")]
public int SpellElements { get; set; } = 0;
```

You must provide a string description for each flag. In this example, `Fire` has value 1, `Water` has value 2, `Earth` has value 4 and `Wind` corresponds to value 8. Usually, constants should be defined accordingly (e.g. `private const int ElementWind = 8` and so on).

You can add explicit values using a colon:

```
[Export(PropertyHint.Flags, "Self:4,Allies:8,Foes:16")]
public int SpellTargets { get; set; } = 0;
```

Only power of 2 values are valid as bit flags options. The lowest allowed value is 1, as 0 means that nothing is selected. You can also add options that are a combination of other flags:

```
[Export(PropertyHint.Flags, "Self:4,Allies:8,Self and
Allies:12,Foes:16")]
public int SpellTargets { get; set; } = 0;
```

Export annotations are also provided for the physics and render layers defined in the project settings.

```
[Export(PropertyHint.Layers2DPhysics)]
public uint Layers2DPhysics { get; set; }
[Export(PropertyHint.Layers2DRender)]
public uint Layers2DRender { get; set; }
[Export(PropertyHint.Layers3DPhysics)]
public uint Layers3DPhysics { get; set; }
[Export(PropertyHint.Layers3DRender)]
public uint Layers3DRender { get; set; }
```

Using bit flags requires some understanding of bitwise operations. If in doubt, use boolean variables instead.

Exporting enums

Members whose type is an enum can be exported and their values are limited to the members of the enum type. The editor will create a widget in the Inspector, enumerating the following as "Thing 1", "Thing 2", "Another Thing". The value will be stored as an integer.

```
public enum MyEnum
{
    Thing1,
    Thing2,
    AnotherThing = -1,
}
```

```
[Export]
public MyEnum MyEnum { get; set; }
```

Integer and string members can also be limited to a specific list of values using the `[Export]` annotation with the `PropertyHint.Enum` hint. The editor will create a widget in the Inspector, enumerating the following as Warrior, Magician, Thief. The value will be stored as an integer, corresponding to the index of the selected option (i.e. `0`, `1`, or `2`).

```
[Export(PropertyHint.Enum, "Warrior,Magician,Thief")]
public int CharacterClass { get; set; };
```

You can add explicit values using a colon:

```
[Export(PropertyHint.Enum, "Slow:30,Average:60,Very Fast:200")]
public int CharacterSpeed { get; set; }
```

If the type is `string`, the value will be stored as a string.

```
[Export(PropertyHint.Enum, "Rebecca,Mary,Leah")]
public string CharacterName { get; set; }
```

If you want to set an initial value, you must specify it explicitly:

```
[Export(PropertyHint.Enum, "Rebecca,Mary,Leah")]
public string CharacterName { get; set; } = "Rebecca";
```

Exporting collections

As explained in the C# Variant documentation, only certain C# arrays and the collection types defined in the `Godot.Collections` namespace are Variant-compatible, therefore, only those types can be exported.

Exporting Godot arrays

```
[Export]
public Godot.Collections.Array Array { get; set; }
```

Using the generic `Godot.Collections.Array<T>` allows specifying the type of the array elements, which will be used as a hint for the editor. The Inspector will restrict the elements to the specified type.

```
[Export]
public Godot.Collections.Array<string> Array { get; set; }
```

The default value of Godot arrays is null. A different default can be specified:

```
[Export]
public Godot.Collections.Array<string> CharacterNames { get;
set; } = new Godot.Collections.Array<string>
{
    "Rebecca",
    "Mary",
    "Leah",
};
```

Arrays with specified types which inherit from resource can be set by drag-and-dropping multiple files from the FileSystem dock.

```
[Export]
public Godot.Collections.Array<Texture> Textures { get; set; }

[Export]
public Godot.Collections.Array<PackedScene> Scenes { get; set;
}
```

Exporting Godot dictionaries

```
[Export]
public Godot.Collections.Dictionary Dictionary { get; set; }
```

Using the generic `Godot.Collections.Dictionary<TKey, TValue>` allows specifying the types of the key and value elements of the dictionary.

Note

Typed dictionaries are currently unsupported in the Godot editor, so the Inspector will not restrict the types that can be assigned, potentially resulting in runtime exceptions.

```
[Export]
public Godot.Collections.Dictionary<string, int> Dictionary {
get; set; }
```

The default value of Godot dictionaries is null. A different default can be specified:

```
[Export]
public Godot.Collections.Dictionary<string, int> CharacterLives
{ get; set; } = new Godot.Collections.Dictionary<string, int>
{
    ["Rebecca"] = 10,
    ["Mary"] = 42,
    ["Leah"] = 0,
};
```

Exporting C# arrays

C# arrays can exported as long as the element type is a [Variant-compatible type].

```
[Export]
public Vector3[] Vectors { get; set; }

[Export]
public NodePath[] NodePaths { get; set; }
```

The default value of C# arrays is null. A different default can be specified:

```
[Export]
public Vector3[] Vectors { get; set; } = new Vector3[]
{
    new Vector3(1, 2, 3),
    new Vector3(3, 2, 1),
}
```

Setting exported variables from a tool script

When changing an exported variable's value from a script in Tool mode, the value in the inspector won't be updated automatically. To update it, call NotifyPropertyListChanged() after setting the exported variable's value.

Advanced exports

Not every type of export can be provided on the level of the language itself to avoid unnecessary design complexity. The following describes some more or less common exporting features which can be implemented with a low-level API.

Before reading further, you should get familiar with the way properties are handled and how they can be customized with _Set(), _Get(), and _GetPropertyList() methods as described in Accessing data or logic from an object.

See also

For binding properties using the above methods in C++, see Binding properties using _set/_get/_get_property_list.

Warning

The script must operate in the `tool` mode so the above methods can work from within the editor.

C# global classes

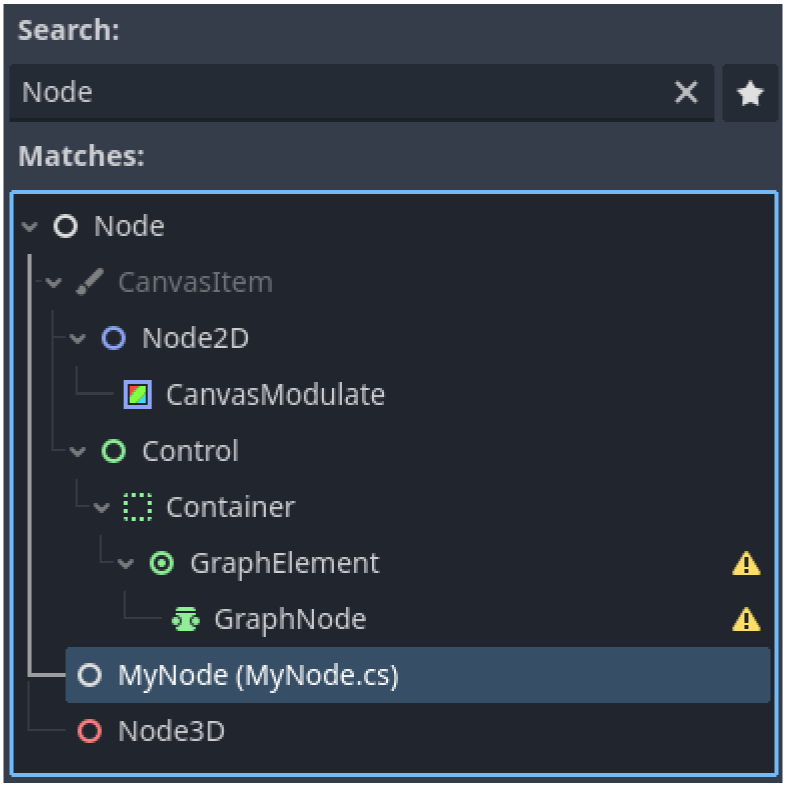
Global classes (also known as named scripts) are types registered in Godot's editor so they can be used more conveniently.

- Global classes show up in the *Add Node* and *Create Resource* dialogs.
- If an exported property is a global class, the inspector restricts assignment, allowing only instances of that global class or any derived classes.

Global classes are registered with the `[GlobalClass]` attribute.

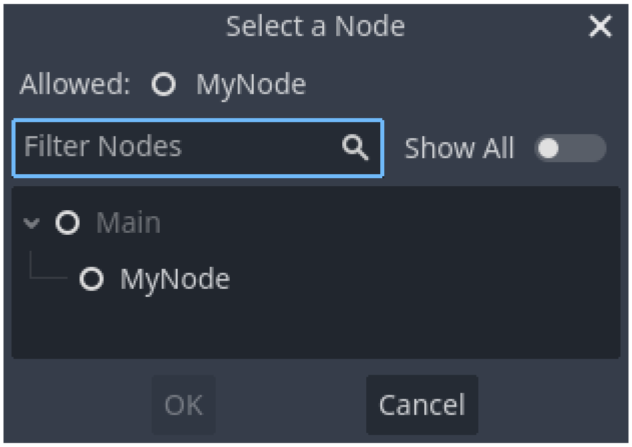
```
using Godot;

[GlobalClass]
public partial class MyNode : Node
{
}
```

The `MyNode` type will be registered as a global class with the same name as the type's name.


Search:
Node
Matches:
Node
CanvasItem
Node2D
CanvasModulate
Control
Container
GraphElement
GraphNode
MyNode (MyNode.cs)
Node3D

The *Select a Node* window for the `MyNode` exported property filters the list of nodes in the scene to match the assignment restriction.

```
public partial class Main : Node
{
    [Export]
    public MyNode MyNode { get; set; }
}
```

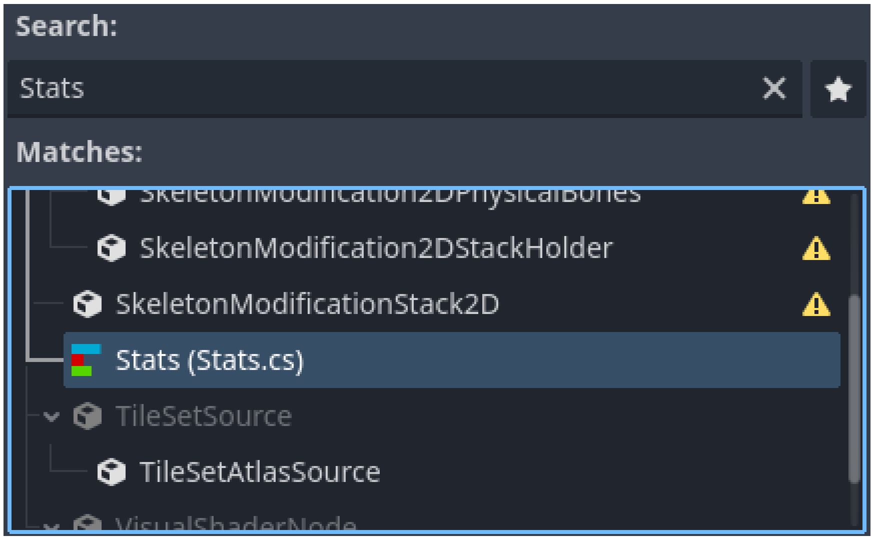

Select a Node
Allowed: MyNode
Filter Nodes
Show All
Main
MyNode
OK
Cancel

If a custom type isn't registered as a global class, the assignment is restricted to the Godot type the custom type is based on. For example, inspector assignments to an export of the type `MySimpleSprite2D` are restricted to `Sprite2D` and derived types.

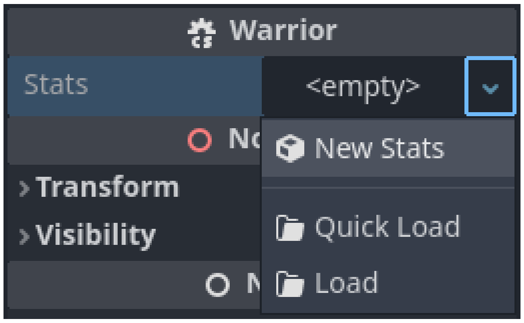
```
public partial class MySimpleSprite2D : Sprite2D
{
}
```

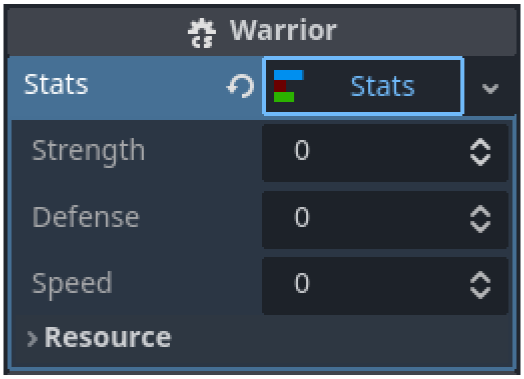
When combined with the `[GlobalClass]` attribute, the `[Icon]` attribute allows providing a path to an icon to show when the class is displayed in the editor.

```
using Godot;

[GlobalClass, Icon("res://Stats/StatsIcon.svg")]
public partial class Stats : Resource
{
    [Export]
    public int Strength { get; set; }

    [Export]
    public int Defense { get; set; }

    [Export]
    public int Speed { get; set; }
}
```


Search:
Stats
Matches:
SkeletonModification2DPhysicalBones
SkeletonModification2DStackHolder
SkeletonModificationStack2D
Stats (Stats.cs)
TileSetSource
TileSetAtlasSource
VisualShaderNode

The `Stats` class is a custom resource registered as a global class. Exporting properties of the type `Stats` will only allow instances of this resource type to be assigned, and the inspector will let you create and load instances of this type easily.


Warrior
Stats
<empty>
New Stats
Quick Load
Load
Transform
Visibility


Warrior
Stats
Stats
Resource

Strength 0
Defense 0
Speed 0

Warning

The Godot editor will hide these custom classes with names that begin with the prefix "Editor" in the 'Create New Node' or 'Create New Scene' dialog windows. The classes are available for instantiation at runtime via their class names, but are automatically hidden by the editor windows along with the built-in editor nodes used by the Godot editor.

GDExtension

- What is GDExtension?
- GDExtension C++ example
- The .gdextension file
- GDExtension documentation system

What is GDExtension?

Introduction

GDExtension is a Godot-specific technology that lets the engine interact with native shared libraries [https://en.wikipedia.org/wiki/Library_(computing)#Shared_libraries] at run-time. You can use it to run native code without compiling it with the engine.

Note

GDExtension is *not* a scripting language and has no relation to GDScript.

Differences between GDExtension and C++ modules

You can use both GDExtension and C++ modules to run C or C++ code in a Godot project.

They also both allow you to integrate third-party libraries into Godot. The one you should choose depends on your needs.

Warning

Our long-term goal is that GDExtensions targeting an earlier version of Godot will work in later minor versions, but not vice-versa. For example, a GDExtension targeting Godot 4.2 should work just fine in Godot 4.3, but one targeting Godot 4.3 won't work in Godot 4.2.

However, GDExtension is currently *experimental*, which means that we may break compatibility in order to fix major bugs or include critical features. For example, GDExtensions created for Godot 4.0 aren't compatible with Godot 4.1 (see Updating your GDExtension for 4.1).

Advantages of GDExtension

Unlike modules, GDExtension doesn't require compiling the engine's source code, making it easier to distribute your work. It gives you access to most of the API available to GDScript and C#, allowing you to code game logic with full control regarding performance. It's ideal if you need high-performance code you'd like to distribute as an add-on in the asset library.

Also:

- GDExtension is not limited to C and C++. Thanks to third-party bindings, you can use it with many other languages.
- You can use the same compiled GDExtension library in the editor and exported project. With C++ modules, you have to recompile all the export templates you plan to use if you require its functionality at run-time.
- GDExtension only requires you to compile your library, not the whole engine. That's unlike C++ modules, which are statically compiled into the engine. Every time you change a module, you need to recompile the engine. Even with incremental builds, this process is slower than using GDExtension.

Advantages of C++ modules

We recommend C++ modules in cases where GDExtension isn't enough:

- C++ modules provide deeper integration into the engine. GDExtension's access is not as deep as static modules.
- You can use C++ modules to provide additional features in a project without carrying native library files around. This extends to exported projects.

Note

If you notice that specific systems are not accessible via GDExtension but are via custom modules, feel free to open an issue on the godot-cpp repository [https://github.com/godotengine/godot-cpp] to discuss implementation options for exposing the missing functionality.

Supported languages

The Godot developers officially support the following language bindings for GDExtension:

- C++ (tutorial)

Note

There are no plans to support additional languages with GDExtension officially. That said, the community offers several bindings for other languages (see below).

The bindings below are developed and maintained by the community:

- D [https://github.com/godot-dlang/godot-dlang]
- Go [https://github.com/grow-graphics/gd]
- Haxe [https://hxgodot.github.io/]
- Rust [https://github.com/godot-rust/gdext]
- Swift [https://github.com/migueldeicaza/SwiftGodot]

Note

Not all bindings mentioned here may be production-ready. Make sure to research options thoroughly before starting a project with one of those. Also, double-check whether the binding is compatible with the Godot version you're using.

Version compatibility

GDExtension add-ons compiled for a given Godot version are only guaranteed to work with the same minor release series. For example, a GDExtension add-on compiled for Godot 4.0 will only work with Godot 4.0, 4.0.1, 4.0.2. In addition, GDExtension is not compatible with Godot 3.x.

GDExtension add-ons are also only compatible with engine builds that use the level of floating-point precision the extension was compiled for. This means that if you use an engine build with double-precision floats, the extension must also be compiled for double-precision floats. See Large world coordinates for details.

GDExtension C++ example

Introduction

The C++ bindings for GDExtension are built on top of the C GDExtension API and provide a nicer way to "extend" nodes and other built-in classes in Godot using C++. This new system allows the extension of Godot to nearly the same level as statically linked C++ modules.

You can download the included example in the test folder of the godot-cpp repository on GitHub [https://github.com/godotengine/godot-cpp].

Setting up the project

There are a few prerequisites you'll need:

- a Godot 4 executable,
- a C++ compiler,
- SCons as a build tool,
- a copy of the godot-cpp repository [https://github.com/godotengine/godot-cpp].

See also Compiling as the build tools are identical to the ones you need to compile Godot from source.

You can download the godot-cpp repository [https://github.com/godotengine/godot-cpp] from GitHub or let Git do the work for you. Note that this repository has different branches for different versions of Godot. GDExtensions will not work in older versions of Godot (only Godot 4 and up) and vice versa, so make sure you download the correct branch.

Note

To use GDExtension [https://godotengine.org/article/introducing-gd-extensions] you need to use the godot-cpp branch that matches the version of Godot that you are targeting. For example, if you're targeting Godot 4.1, use the `4.1` branch. Throughout this tutorial we use `4.x`, which will need to be replaced with the version of Godot you are targeting.

The `master` branch is the development branch which is updated regularly to work with Godot's `master` branch.

Warning

Our long-term goal is that GDExtensions targeting an earlier version of Godot will work in later minor versions, but not vice-versa. For example, a GDExtension targeting Godot 4.1 should work just fine in Godot 4.2, but one targeting Godot 4.2 won't work in Godot 4.1.

However, GDExtension is currently *experimental*, which means that we may break compatibility in order to fix major bugs or include critical features. For example, GDExtensions created for Godot 4.0 aren't compatible with Godot 4.1 (see Updating your GDExtension for 4.1).

If you are versioning your project using Git, it is recommended to add it as a Git submodule:

```
mkdir gdextension_cpp_example
cd gdextension_cpp_example
git init
git submodule add -b 4.x https://github.com/godotengine/godot-
cpp
cd godot-cpp
git submodule update --init
```

Alternatively, you can also clone it to the project folder:

```
mkdir gdextension_cpp_example
cd gdextension_cpp_example
git clone -b 4.x https://github.com/godotengine/godot-cpp
```

Note

If you decide to download the repository or clone it into your folder, make sure to keep the folder layout the same as we've setup here. Much of the code we'll be showcasing here assumes the project has this layout.

If you cloned the example from the link specified in the introduction, the submodules are not automatically initialized. You will need to execute the following commands:

```
cd gdextension_cpp_example
git submodule update --init
```

This will initialize the repository in your project folder.

Building the C++ bindings

Now that we've downloaded our prerequisites, it is time to build the C++ bindings.

The repository contains a copy of the metadata for the current Godot release, but if you need to build these bindings for a newer version of Godot, simply call the Godot executable:

```
godot --dump-extension-api
```

The resulting `extension_api.json` file will be created in the executable's directory. Copy it to the project folder and add `custom_api_file=<PATH_TO_FILE>` to the scons command below.

To generate and compile the bindings, use this command (replacing `<platform>` with `windows`, `linux` or `macos` depending on your OS):

The build process automatically detects the number of CPU threads to use for parallel builds. To specify a number of CPU threads to use, add `-jN` at the end of the SCons command line where `N` is the number of CPU threads to use.

```
cd godot-cpp
scons platform=<platform> custom_api_file=<PATH_TO_FILE>
cd ..
```

This step will take a while. When it is completed, you should have static libraries that can be compiled into your project stored in `godot-cpp/bin/`.

Note

You may need to add `bits=64` to the command on Windows or Linux.

Creating a simple plugin

Now it's time to build an actual plugin. We'll start by creating an empty Godot project in which we'll place a few files.

Open Godot and create a new project. For this example, we will place it in a folder called `demo` inside our GDExtension's folder structure.

In our demo project, we'll create a scene containing a Node called "Main" and we'll save it as `main.tscn`. We'll come back to that later.

Back in the top-level GDExtension module folder, we're also going to create a subfolder called `src` in which we'll place our source files.

You should now have `demo`, `godot-cpp`, and `src` directories in your GDExtension module.

Your folder structure should now look like this:

```
gdextension_cpp_example/
|
+--demo/                  # game example/demo to test the
extension
|
+--godot-cpp/             # C++ bindings
|
+--src/                   # source code of the extension we are
building
```

In the `src` folder, we'll start with creating our header file for the GDExtension node we'll be creating. We will name it `gdexample.h`:

```
#ifndef GDEXAMPLE_H
#define GDEXAMPLE_H

#include <godot_cpp/classes/sprite2d.hpp>

namespace godot {

class GDExample : public Sprite2D {
    GDCLASS(GDExample, Sprite2D)

private:
    double time_passed;

protected:
    static void _bind_methods();

public:
    GDExample();
    ~GDExample();

    void _process(double delta) override;
};

}

#endif
```

There are a few things of note to the above. We include `sprite2d.hpp` which contains bindings to the Sprite2D class. We'll be extending this class in our module.

We're using the namespace `godot`, since everything in GDExtension is defined within this namespace.

Then we have our class definition, which inherits from our Sprite2D through a container class. We'll see a few side effects of this later on. The `GDCLASS` macro sets up a few internal things for us.

After that, we declare a single member variable called `time_passed`.

In the next block we're defining our methods, we have our constructor and destructor defined, but there are two other functions that will likely look familiar to some, and one new method.

The first is `_bind_methods`, which is a static function that Godot will call to find out which methods can be called and which properties it exposes. The second is our `_process` function, which will work exactly the same as the `_process` function you're used to in GDScript.

Let's implement our functions by creating our `gdexample.cpp` file:

```
#include "gdexample.h"
#include <godot_cpp/core/class_db.hpp>
```

```
using namespace godot;

void GDExample::_bind_methods() {
}

GDExample::GDExample() {
    // Initialize any variables here.
    time_passed = 0.0;
}

GDExample::~GDExample() {
    // Add your cleanup here.
}

void GDExample::_process(double delta) {
    time_passed += delta;

    Vector2 new_position = Vector2(10.0 + (10.0 *
sin(time_passed * 2.0)), 10.0 + (10.0 * cos(time_passed *
1.5)));

    set_position(new_position);
}
```

This one should be straightforward. We're implementing each method of our class that we defined in our header file.

Note our `_process` function, which keeps track of how much time has passed and calculates a new position for our sprite using a sine and cosine function.

There is one more C++ file we need; we'll name it `register_types.cpp`. Our GDExtension plugin can contain multiple classes, each with their own header and source file like we've implemented `GDExample` up above. What we need now is a small bit of code that tells Godot about all the classes in our GDExtension plugin.

```
#include "register_types.h"
```

```
#include "gdexample.h"

#include <gdextension_interface.h>
#include <godot_cpp/core/defs.hpp>
#include <godot_cpp/godot.hpp>

using namespace godot;

void initialize_example_module(ModuleInitializationLevel
p_level) {
    if (p_level != MODULE_INITIALIZATION_LEVEL_SCENE) {
        return;
    }

    GDREGISTER_CLASS(GDExample);
}

void uninitialize_example_module(ModuleInitializationLevel
p_level) {
    if (p_level != MODULE_INITIALIZATION_LEVEL_SCENE) {
        return;
    }
}

extern "C" {
// Initialization.
GDExtensionBool GDE_EXPORT
example_library_init(GDExtensionInterfaceGetProcAddress
p_get_proc_address, const GDExtensionClassLibraryPtr p_library,
GDExtensionInitialization *r_initialization) {
    godot::GDExtensionBinding::InitObject
init_obj(p_get_proc_address, p_library, r_initialization);

    init_obj.register_initializer(initialize_example_module);
    init_obj.register_terminator(uninitialize_example_module);

init_obj.set_minimum_library_initialization_level(MODULE_INITIA
LIZATION_LEVEL_SCENE);

    return init_obj.init();
}
}
```

The `initialize_example_module` and `uninitialize_example_module` functions get called respectively when Godot loads our plugin and when it unloads it. All we're doing here is parse through the functions in our bindings module to initialize them, but you might have to set up more things depending on your needs. We call the `GDREGISTER_CLASS` macro for each of our classes in our library.

The important function is the third function called `example_library_init`. We first call a function in our bindings library that creates an initialization object. This object registers the initialization and termination functions of the GDExtension. Furthermore, it sets the level of initialization (core, servers, scene, editor, level).

At last, we need the header file for the `register_types.cpp` named `register_types.h`.

```
#ifndef GDEXAMPLE_REGISTER_TYPES_H
#define GDEXAMPLE_REGISTER_TYPES_H

#include <godot_cpp/core/class_db.hpp>

using namespace godot;

void initialize_example_module(ModuleInitializationLevel
p_level);
void uninitialize_example_module(ModuleInitializationLevel
p_level);

#endif // GDEXAMPLE_REGISTER_TYPES_H
```

Compiling the plugin

We cannot easily write by hand a `SConstruct` file that SCons would use for building. For the purpose of this example, just use **`this hardcoded SConstruct file`** we've prepared. We'll cover a more customizable, detailed example on how to use these build files in a subsequent tutorial.

Note

This `SConstruct` file was written to be used with the latest `godot-cpp` master, you may need to make small changes using it with older versions or refer to the `SConstruct` file in the Godot 4.x documentation.

Once you've downloaded the `SConstruct` file, place it in your GDExtension folder structure alongside `godot-cpp`, `src` and `demo`, then run:

```
scons platform=<platform>
```

You should now be able to find the module in `demo/bin/<platform>`.

When building for iOS, package the module as a static *.xcframework*, you can use following commands to do so:

```
# compile simulator and device modules
scons arch=universal ios_simulator=yes platform=ios target=<target>
scons arch=arm64 ios_simulator=no platform=ios target=<target>

# assemble xcframeworks
xcodebuild -create-xcframework -library demo/bin/libgdexample.ios.<target>.a -library demo/bin/libgdexample.ios.<target>.simulator.a -output demo/bin/libgdexample.ios.<target>.xcframework
xcodebuild -create-xcframework -library godot-cpp/bin/libgodot-cpp.ios.<target>.arm64.a -library godot-cpp/bin/libgodot-cpp.ios.<target>.universal.simulator.a  -output demo/bin/libgodot-cpp.ios.<target>.xcframework
```

Note

Here, we've compiled both godot-cpp and our gdexample library as debug builds. For optimized builds, you should compile them using the `target=template_release` switch.

Using the GDExtension module

Before we jump back into Godot, we need to create one more file in `demo/bin/`.

This file lets Godot know what dynamic libraries should be loaded for each platform and the entry function for the module. It is called `gdexample.gdextension`.

```
[configuration]

entry_symbol = "example_library_init"
compatibility_minimum = "4.1"
reloadable = true

[libraries]

macos.debug =
"res://bin/libgdexample.macos.template_debug.framework"
macos.release =
"res://bin/libgdexample.macos.template_release.framework"
ios.debug =
"res://bin/libgdexample.ios.template_debug.xcframework"
ios.release =
"res://bin/libgdexample.ios.template_release.xcframework"
windows.debug.x86_32 =
"res://bin/libgdexample.windows.template_debug.x86_32.dll"
windows.release.x86_32 =
"res://bin/libgdexample.windows.template_release.x86_32.dll"
windows.debug.x86_64 =
"res://bin/libgdexample.windows.template_debug.x86_64.dll"
windows.release.x86_64 =
"res://bin/libgdexample.windows.template_release.x86_64.dll"
linux.debug.x86_64 =
"res://bin/libgdexample.linux.template_debug.x86_64.so"
linux.release.x86_64 =
"res://bin/libgdexample.linux.template_release.x86_64.so"
linux.debug.arm64 =
"res://bin/libgdexample.linux.template_debug.arm64.so"
linux.release.arm64 =
"res://bin/libgdexample.linux.template_release.arm64.so"
linux.debug.rv64 =
"res://bin/libgdexample.linux.template_debug.rv64.so"
linux.release.rv64 =
"res://bin/libgdexample.linux.template_release.rv64.so"
android.debug.x86_64 =
"res://bin/libgdexample.android.template_debug.x86_64.so"
android.release.x86_64 =
"res://bin/libgdexample.android.template_release.x86_64.so"
android.debug.arm64 =
"res://bin/libgdexample.android.template_debug.arm64.so"
android.release.arm64 =
"res://bin/libgdexample.android.template_release.arm64.so"

[dependencies]
ios.debug = {
    "res://bin/libgodot-cpp.ios.template_debug.xcframework": ""
}
ios.release = {
    "res://bin/libgodot-cpp.ios.template_release.xcframework":
""
}
```

This file contains a `configuration` section that controls the entry function of the module. You should also set the minimum compatible Godot version with `compatability_minimum`, which prevents older version of Godot from trying to load your extension. The `reloadable` flag enables automatic reloading of your extension by the editor every time you recompile it, without needing to restart the editor. This only works if you compile your extension in debug mode (default).

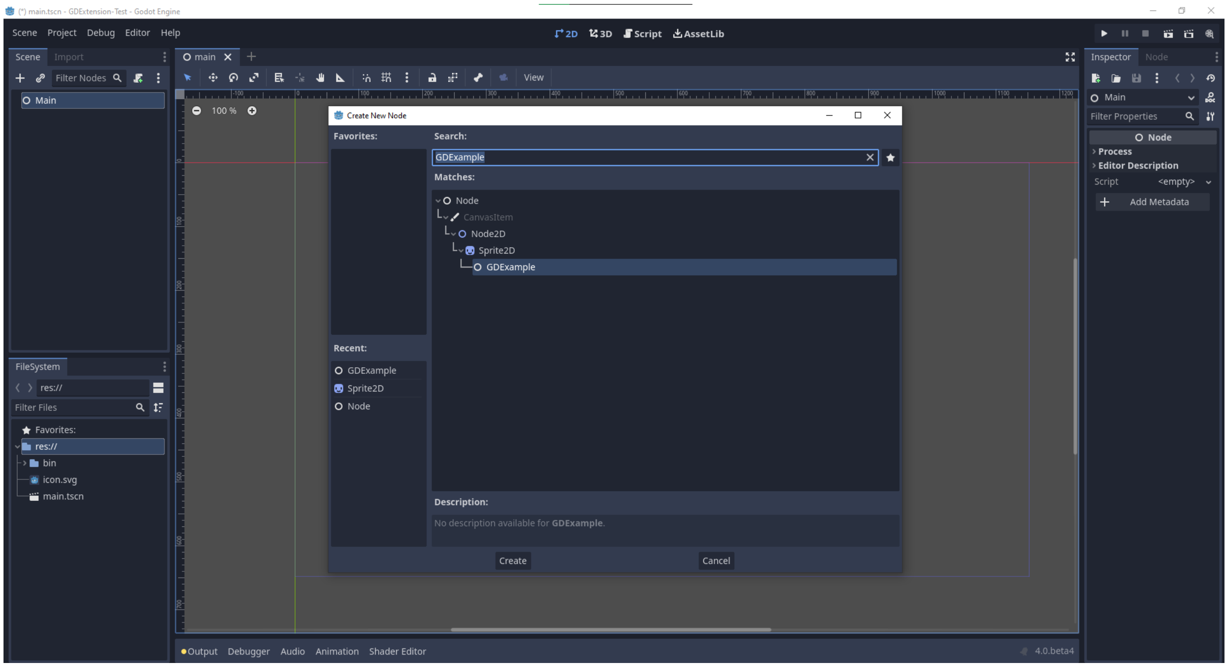
The `libraries` section is the important bit: it tells Godot the location of the dynamic library in the project's filesystem for each supported platform. It will also result in *just* that file being exported when you export the project, which means the data pack won't contain libraries that are incompatible with the target platform.

Finally, the `dependencies` section allows you to name additional dynamic libraries that should be included as well. This is important when your GDExtension plugin implements someone else's library and requires you to supply a third-party dynamic library with your project.

Here is another overview to check the correct file structure:

```
gdextension_cpp_example/
|
+--demo/                  # game example/demo to test the 
extension
|   |
|   +--main.tscn
|   |
|   +--bin/
|       |
|       +--gdexample.gdextension
|
+--godot-cpp/             # C++ bindings
|
+--src/                   # source code of the extension we are 
building
|   |
|   +--register_types.cpp
|   +--register_types.h
|   +--gdexample.cpp
|   +--gdexample.h
```

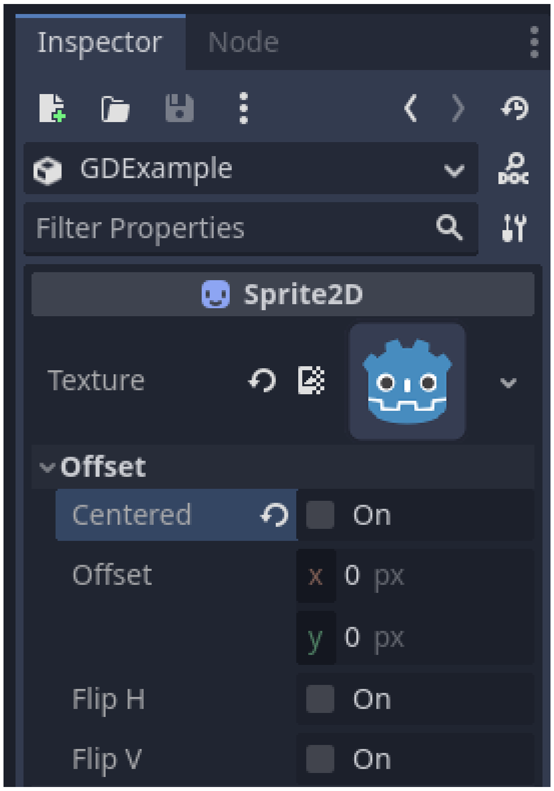
Time to jump back into Godot. We load up the main scene we created way back in the beginning and now add a newly available GDExample node to the scene:


(*) main.tscn - GDExtension-Test - Godot Engine
Scene Project Debug Editor Help
2D 3D Script AssetLib
Create New Node
Favorites:
Search:
Matches:
Node
CanvasItem
Node2D
Sprite2D
GDExample
Recent:
GDExample
Sprite2D
Node
Create
Cancel
Inspector
Node
Main
FileSystem
res://
bin
icon.svg
main.tscn
Output Debugger Audio Animation Shader Editor
4.0.beta4

Search: GDExample

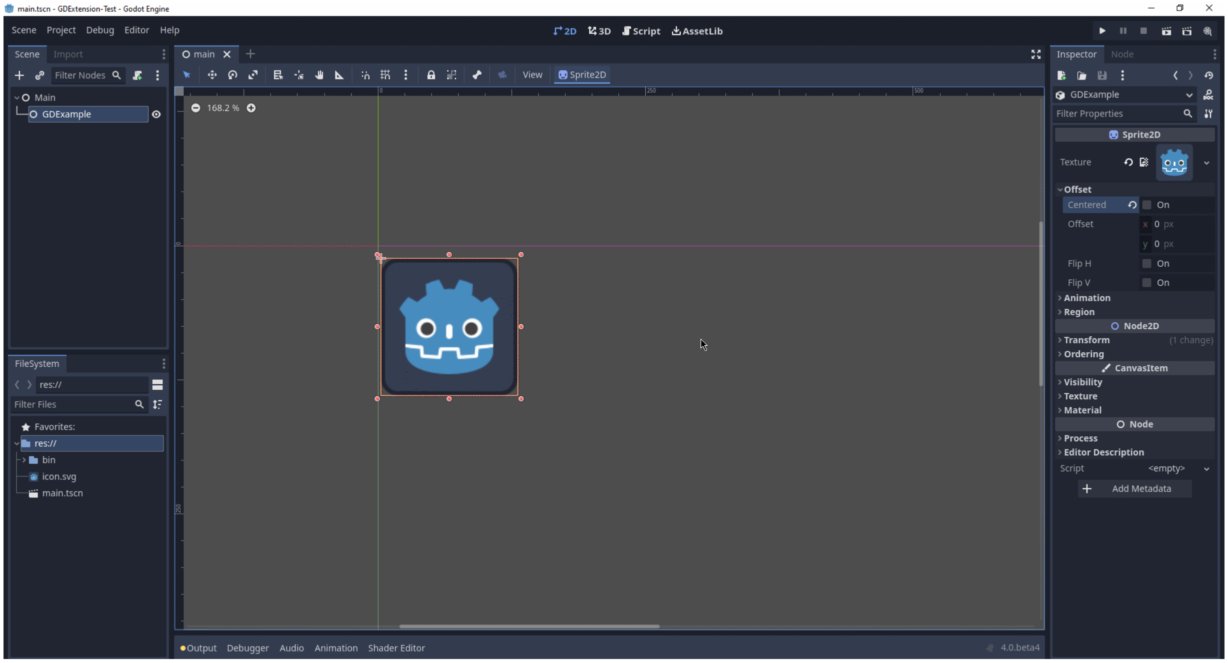
Description: No description available for GDExample.

We're going to assign the Godot logo to this node as our texture, disable the `centered` property:


Inspector
Node
GDExample
Filter Properties
Sprite2D
Texture
Offset

Centered ☐ On
Offset x 0 px
y 0 px
Flip H ☐ On
Flip V ☐ On

We're finally ready to run the project:


main.tscn - GDExtension-Test - Godot Engine
Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Scene
Import
Filter Nodes
Main
GDExample
main
View
Sprite2D
168.2 %
Inspector
Node
GDExample
Filter Properties
Sprite2D
Texture
Offset
Centered
On
Offset
x 0 px
y 0 px
Flip H
On
Flip V
On
Animation
Region
Node2D
Transform
(1 change)
Ordering
CanvasItem
Visibility
Texture
Material
Node
Process
Editor Description
Script
<empty>
Add Metadata
FileSystem
res://
Filter Files
Favorites:
res://
bin
icon.svg
main.tscn
Output
Debugger
Audio
Animation
Shader Editor
4.0.beta4

Adding properties

GDScript allows you to add properties to your script using the `export` keyword. In GDExtension you have to register the properties with a getter and setter function or directly implement the `_get_property_list`, `_get` and `_set` methods of an object (but that goes far beyond the scope of this tutorial).

Lets add a property that allows us to control the amplitude of our wave.

In our `gdexample.h` file we need to add a member variable and getter and setter functions:

```
...
private:
    double time_passed;
    double amplitude;

public:
    void set_amplitude(const double p_amplitude);
    double get_amplitude() const;
...
```

In our `gdexample.cpp` file we need to make a number of changes, we will only show the methods we end up changing, don't remove the lines we're omitting:

```
void GDExample::_bind_methods() {
    ClassDB::bind_method(D_METHOD("get_amplitude"),
&GDExample::get_amplitude);
    ClassDB::bind_method(D_METHOD("set_amplitude",
"p_amplitude"), &GDExample::set_amplitude);

    ADD_PROPERTY(PropertyInfo(Variant::FLOAT, "amplitude"),
"set_amplitude", "get_amplitude");
}

GDExample::GDExample() {
    // Initialize any variables here.
    time_passed = 0.0;
    amplitude = 10.0;
}

void GDExample::_process(double delta) {
    time_passed += delta;

    Vector2 new_position = Vector2(
        amplitude + (amplitude * sin(time_passed * 2.0)),
        amplitude + (amplitude * cos(time_passed * 1.5))
    );

    set_position(new_position);
}

void GDExample::set_amplitude(const double p_amplitude) {
    amplitude = p_amplitude;
}

double GDExample::get_amplitude() const {
    return amplitude;
}
```

Once you compile the module with these changes in place, you will see that a property has been added to our interface. You can now change this property and when you run your project, you will see that our Godot icon travels along a larger figure.

Let's do the same but for the speed of our animation and use a setter and getter function. Our `gdexample.h` header file again only needs a few more lines of code:

```
...
    double amplitude;
    double speed;
...
    void _process(double delta) override;
    void set_speed(const double p_speed);
    double get_speed() const;
...
```

This requires a few more changes to our `gdexample.cpp` file, again we're only showing the methods that have changed so don't remove anything we're omitting:

```
void GDExample::_bind_methods() {
    ...
    ClassDB::bind_method(D_METHOD("get_speed"),
&GDExample::get_speed);
    ClassDB::bind_method(D_METHOD("set_speed", "p_speed"),
&GDExample::set_speed);

    ADD_PROPERTY(PropertyInfo(Variant::FLOAT, "speed",
PROPERTY_HINT_RANGE, "0,20,0.01"), "set_speed", "get_speed");
}

GDExample::GDExample() {
    time_passed = 0.0;
    amplitude = 10.0;
    speed = 1.0;
}

void GDExample::_process(double delta) {
    time_passed += speed * delta;

    Vector2 new_position = Vector2(
        amplitude + (amplitude * sin(time_passed * 2.0)),
        amplitude + (amplitude * cos(time_passed * 1.5))
    );

    set_position(new_position);
}

...

void GDExample::set_speed(const double p_speed) {
    speed = p_speed;
}

double GDExample::get_speed() const {
    return speed;
}
```

Now when the project is compiled, we'll see another property called speed. Changing its value will make the animation go faster or slower. Furthermore, we added a property range which describes in which range the value can be. The first two arguments are the minimum and maximum value and the third is the step size.

Note

For simplicity, we've only used the hint_range of the property method. There are a lot more options to choose from. These can be used to further configure how properties are displayed and set on the Godot side.

Signals

Last but not least, signals fully work in GDExtension as well. Having your extension react to a signal given out by another object requires you to call `connect` on that object. We can't think of a good example for our wobbling Godot icon, we would need to showcase a far more complete example.

This is the required syntax:

```
some_other_node->connect("the_signal", Callable(this,
"my_method"));
```

To connect our signal `the_signal` from some other node with our method `my_method`, we need to provide the `connect` method with the name of the signal and a `Callable`. The `Callable` holds information about an object on which a method can be called. In our case, it associates our current object instance `this` with the method `my_method` of the object. Then the `connect` method will add this to the observers of `the_signal`. Whenever `the_signal` is now emitted, Godot knows which method of which object it needs to call.

Note that you can only call `my_method` if you've previously registered it in your `_bind_methods` method. Otherwise Godot will not know about the existence of `my_method`.

To learn more about `Callable`, check out the class reference here: Callable.

Having your object sending out signals is more common. For our wobbling Godot icon, we'll do something silly just to show how it works. We're going to emit a signal every time a second has passed and pass the new location along.

In our `gdexample.h` header file, we need to define a new member `time_emit`:

```
...
    double time_passed;
    double time_emit;
    double amplitude;
...
```

This time, the changes in `gdexample.cpp` are more elaborate. First, you'll need to set `time_emit = 0.0;` in either our `_init` method or in our constructor. We'll look at the other 2 needed changes one by one.

In our `_bind_methods` method, we need to declare our signal. This is done as follows:

```
void GDExample::_bind_methods() {
    ...
    ADD_PROPERTY(PropertyInfo(Variant::FLOAT, "speed",
PROPERTY_HINT_RANGE, "0,20,0.01"), "set_speed", "get_speed");

    ADD_SIGNAL(MethodInfo("position_changed",
PropertyInfo(Variant::OBJECT, "node"),
PropertyInfo(Variant::VECTOR2, "new_pos")));
}
```

Here, our `ADD_SIGNAL` macro can be a single call with a `MethodInfo` argument. `MethodInfo`'s first parameter will be the signal's name, and its remaining parameters are `PropertyInfo` types which describe the essentials of each of the method's parameters. `PropertyInfo` parameters are defined with the data type of the parameter, and then the name that the parameter will have by default.

So here, we add a signal, with a `MethodInfo` which names the signal "position_changed". The `PropertyInfo` parameters describe two essential arguments, one of type `Object`, the other of type `Vector2`, respectively named "node" and "new_pos".

Next, we'll need to change our `_process` method:

```
void GDExample::_process(double delta) {
    time_passed += speed * delta;
    Vector2 new_position = Vector2(
        amplitude + (amplitude * sin(time_passed * 2.0)),
        amplitude + (amplitude * cos(time_passed * 1.5))
    );

    set_position(new_position);

    time_emit += delta;
    if (time_emit > 1.0) {
        emit_signal("position_changed", this, new_position);

        time_emit = 0.0;
    }
}
```

After a second has passed, we emit our signal and reset our counter. We can add our parameter values directly to `emit_signal`.

Once the GDExtension library is compiled, we can go into Godot and select our sprite node. In the **Node** dock, we can find our new signal and link it up by pressing the **Connect** button or double-clicking the signal. We've added a script on our main node and implemented our signal like this:

```
extends Node

func _on_Sprite2D_position_changed(node, new_pos):
    print("The position of " + node.get_class() + " is now " +
str(new_pos))
```

Every second, we output our position to the console.

Next steps

We hope the above example showed you the basics. You can build upon this example to create full-fledged scripts to control nodes in Godot using C++.

The .gdextension file

Introduction

The `.gdextension` file in your project contains the instructions for how to load the GDExtension. The instructions are separated into specific sections. This page should give you a quick overview of the different options available to you. For an introduction how to get started with GDExtensions take a look at the GDExtension C++ Example.

Configuration section

Property	Type	Description
entry_symbol	String	Name of the entry function for initializing the GDExtension. This function should be defined in the `register_types.cpp` file when using godot-cpp. Adding this is necessary for the extension to work.
compatibility_minimum	String	Minimum compatible version. This prevents older versions of Godot from loading extensions that depend on features from

Property	Type	Description
		newer versions of Godot. **Only supported in Godot 4.1 or later**
compatibility_maximum	String	Maximum compatible version. This prevents newer versions of Godot from loading the extension. **Only supported in Godot 4.3 or later**
reloadable	Boolean	Reloads the extension upon recompilation. Reloading is supported for the godot-cpp binding in Godot 4.2 or later. Other language bindings may or may not support it as well. This flag should be mainly used for developing or debugging an extension.
android_aar_plugin	Boolean	The GDExtension is part of a v2 Android plugin. During export this flag will indicate to the editor that the GDExtension native shared libraries are exported by the Android plugin AAR binaries.

Libraries section

In this section you can set the paths to the compiled binaries of your GDExtension libraries. By specifying feature flags you can filter which version should be loaded and exported with your game depending on which feature flags are active. Every feature flag must match to Godot's feature flags or your custom export flags to be loaded in an exported game. For instance `macos.debug` means that it will be loaded if Godot has both the `macos` and `debug` flag active. Each line of the section is evaluated from top to bottom.

Here is an example of what that can look like:

```
[libraries]

macos.debug =
"res://bin/libgdexample.macos.template_debug.framework"
macos.release =
"res://bin/libgdexample.macos.template_release.framework"
windows.debug.x86_32 =
"res://bin/libgdexample.windows.template_debug.x86_32.dll"
windows.release.x86_32 =
"res://bin/libgdexample.windows.template_release.x86_32.dll"
windows.debug.x86_64 =
"res://bin/libgdexample.windows.template_debug.x86_64.dll"
windows.release.x86_64 =
"res://bin/libgdexample.windows.template_release.x86_64.dll"
linux.debug.x86_64 =
"res://bin/libgdexample.linux.template_debug.x86_64.so"
linux.release.x86_64 =
"res://bin/libgdexample.linux.template_release.x86_64.so"
linux.debug.arm64 =
"res://bin/libgdexample.linux.template_debug.arm64.so"
linux.release.arm64 =
"res://bin/libgdexample.linux.template_release.arm64.so"
linux.debug.rv64 =
"res://bin/libgdexample.linux.template_debug.rv64.so"
linux.release.rv64 =
"res://bin/libgdexample.linux.template_release.rv64.so"
```

Here are lists of some of the available built-in options (for more look at the feature tags):

Running system

Flag	Description
windows	Windows operating system
macos	Mac operating system
linux	Linux operating system
bsd	BSD operating system
linuxbsd	Linux or BSD operating system
android	Android operating system
ios	iOS operating system
web	Web browser

Build

Flag	Description
debug	Build with debug symbols
release	Optimized build without debug symbols
editor	Editor build

Architecture

Flag	Description
double	double-precision build
single	single-precision build
x86_64	64-bit x86 build
arm64	64-bit ARM build
rv64	64-bit RISC-V build
riscv	RISC-V build (any bitness)

Flag	Description
wasm32	32-bit WebAssembly build

Icons section

By default, Godot uses the Node icon in the scene dock for GDExtension nodes. A custom icon can be set by reference to its name and resource path of an SVG file.

For example:

```
[icons]

GDExample = "res://icons/gd_example.svg"
```

The path should point to a 16 by 16 pixel SVG image. Read the guide for creating icons for more information.

Dependencies section

In this section you set the paths of the GDExtension dependencies. This is used internally to export the dependencies when exporting your game executable. You are able to set which dependency is loaded depending on the feature flags of the exported executable. In addition, you are able to set an optional subdirectory to move your dependencies into. If no path is supplied Godot will move the libraries into the same directory as your game executable.

Warning

In MacOS it is necessary to have shared libraries inside a folder called `Frameworks` with a directory structure like this: `Game.app/Contents/Frameworks`.

```
[dependencies]

macos.debug = {
    "res://bin/libdependency.macos.template_debug.framework" :
"Contents/Frameworks"
}
macos.release = {
    "res://bin/libdependency.macos.template_release.framework"
: "Contents/Frameworks"
}
windows.debug = {
    "res://bin/libdependency.windows.template_debug.x86_64.dll"
: "",
    "res://bin/libdependency.windows.template_debug.x86_32.dll"
: ""
}
windows.release = {

"res://bin/libdependency.windows.template_release.x86_64.dll" :
"",

"res://bin/libdependency.windows.template_release.x86_32.dll" :
""
}
linux.debug = {
    "res://bin/libdependency.linux.template_debug.x86_64.so" :
"",
    "res://bin/libdependency.linux.template_debug.arm64.so" :
"",
    "res://bin/libdependency.linux.template_debug.rv64.so" : ""
}
linux.release = {
    "res://bin/libdependency.linux.template_release.x86_64.so"
: "",
    "res://bin/libdependency.linux.template_release.arm64.so" :
"",
    "res://bin/libdependency.linux.template_release.rv64.so" :
```

```
" "

}
```

GDExtension documentation system

Note

Adding documentation for GDExtensions is only possible for Godot 4.3 and later. The support can be integrated into your project regardless because the snippet will check if you use the appropriate godot-cpp version. If you set the `compatability_minimum` to 4.2 and you load a project with the extension through a 4.2 editor, the documentation page for that class will be empty. The extension itself will still work.

The GDExtension documentation system works in a similar manner to the built-in engine documentation. It uses a series of XML files (one per class) to document the exposed constructors, properties, methods, constants, signals, and theme items of each class.

Note

We are assuming you are using the project files explained in the GDExtension C++ Example with the following structure:

```
gdextension_cpp_example/  # GDExtension directory
|
+--demo/                  # game example/demo to test the
extension
|   |
|   +--main.tscn
|   |
|   +--bin/
|       |
|       +--gdexample.gdextension
|
+--godot-cpp/             # C++ bindings
|
+--src/                   # source code of the extension we are
building
|   |
|   +--register_types.cpp
|   +--register_types.h
|   +--gdexample.cpp
|   +--gdexample.h
```

Inside the Godot demo project directory of your GDExtension directory, run the following terminal command:

```
# Replace "godot" with the full path to a Godot editor binary
# if Godot is not installed in your `PATH`.
godot --doctool ../ --gdextension-docs
```

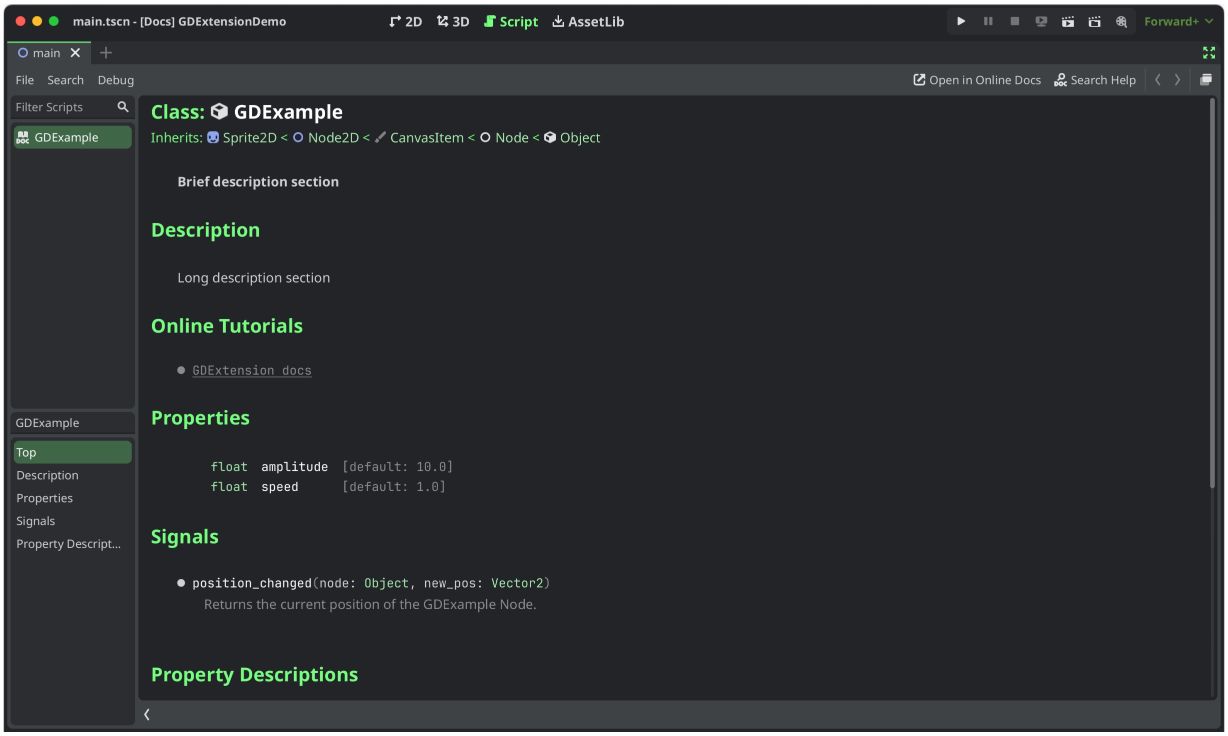
This command calls upon the Godot editor binary to generate documentation via the `--doctool` and `--gdextension-docs` commands. The `../` addition is to let Godot know where the GDExtension SConstruct file is located. By calling this command, Godot generates a `doc_classes` directory inside the project directory in which it generates XML files for the GDExtension classes. Those files can then be edited to add information about member variables, methods, signals, and more.

To add the now edited documentation to the GDExtension and let the editor load it, you need to add the following lines to your SConstruct file:

```
if env["target"] in ["editor", "template_debug"]:
try:
    doc_data = env.GodotCPPDocData("src/gen/doc_data.gen.cpp",
source=Glob("doc_classes/*.xml"))
    sources.append(doc_data)
except AttributeError:
    print("Not including class reference as we're targeting a pre-4.3 baseline.")
```

The if-statement checks if we are compiling the GDExtension library with the `editor` and `template_debug` flags. SCons then tries to load all the XML files inside the `doc_classes` directory and appends them to the `sources` variable which already includes all the source files of your extension. If it fails it means we are currently trying to compile the library when the `godot_cpp` is set to a version before 4.3.

After loading the extension in a 4.3 Godot editor or later and open the documentation of your extension class either by `Ctrl + Click` in the script editor or the Editor help dialog you will see something like this:


main.tscn - [Docs] GDExtensionDemo
2D
3D
Script
AssetLib
Forward+
main
File
Search
Debug
Open in Online Docs
Search Help
Filter Scripts
GDExample
GDExample
Top
Description
Properties
Signals
Property Descript...

Class: GDExample

Inherits: Sprite2D < Node2D < CanvasItem < Node < Object

Brief description section

Description

Long description section

Online Tutorials

- GDExtension docs

Properties

```
float  amplitude  [default: 10.0]
float  speed      [default: 1.0]
```

Signals

- position_changed(node: Object, new_pos: Vector2)
  Returns the current position of the GDExample Node.

Property Descriptions

Documentation styling

To style specific parts of text you can use BBCode tags similarly to how they can be used in RichTextLabels. You can set text as bold, italic, underlined, colored, codeblocks etc. by embedding them in tags like this:

```
[b]this text will be shown as bold[/b]
```

Currently they supported tags for the GDExtension documentation system are:

Tag	Example
b Makes `{text}` use the bold (or bold italics) font of `RichTextLabel`.	`[b]{text}[/b]`
i Makes `{text}` use the italics (or bold italics) font of `RichTextLabel`.	`[i]{text}[/i]`
u Makes `{text}` underlined.	`[u]{text}[/u]`
s Makes `{text}` strikethrough.	`[s]{text}[/s]`
kbd Makes `{text}` use a grey beveled background, indicating a keyboard shortcut.	`[kbd]{text}[/kbd]`

code Makes inline `{text}` use the mono font and styles the text color and background like code.	`[code]{text}[/code]`

codeblocks
Makes multiline `{text}` use the mono font and styles the text color and background like code.
The addition of the `[gdscript]` tag highlights the GDScript specific syntax.
	`[codeblocks]`
	`[gdscript]`
	`{text}`
	`[/gdscript]`
	`[/codeblocks]`

center Makes `{text}` horizontally centered. Same as `[p align=center]`.	`[center]{text}[/center]`
url Creates a hyperlink (underlined and clickable text). Can contain optional `{text}` or display `{link}` as is.	`[url]{link}[/url]` `[url={link}]{text}[/url]`

img
Inserts an image from the `{path}` (can be any valid Texture2D resource).
If `{width}` is provided, the image will try to fit that width maintaining the aspect ratio.
If both `{width}` and `{height}` are provided, the image will be scaled to that size.
	`[img]{path}[/img]`
	`[img={width}]{path}[/img]`
	`[img={width}x{height}]{path}[/img]`
	`[img={valign}]{path}[/img]`
	`[img {options}]{path}[/img]`

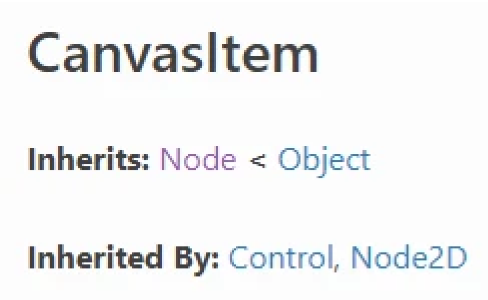
Add `%` to the end of `{width}` or `{height}` value to specify it as percentages of the control width instead of pixels. If `{valign}` configuration is provided, the image will try to align to the surrounding text, see Image and table vertical alignment. Supports configuration options, see Image options.	
color Changes the color of `{text}`. Color must be provided by a common name (see Named colors) or using the HEX format (e.g. `#ff00ff`, see Hexadecimal color codes).	`[color={code/name}]{text}[/color]`

How to read the Godot API

On this page, you'll learn how to read the class reference for the Godot API.

The API, or Application Programming Interface, is an index of what Godot offers users. It provides a brief summary of which classes exist, how they are related to each other, what features they have, and how to use them.

Inheritance



CanvasItem

Inherits: Node < Object

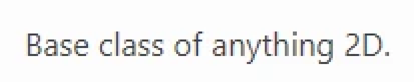
Inherited By: Control, Node2D

At the top of each file, you will see the name of the class.

The "Inherits" section lists each class the current one inherits. Here `CanvasItem` inherits `Node` and `Node` inherits `Object`.

The "Inherited By" section lists each class which directly inherits the current class. Here `Control` and `Node2D` both inherit `CanvasItem`.

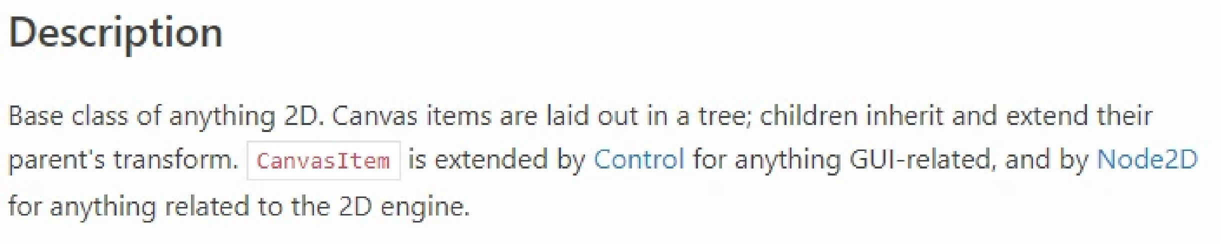
Brief Description



Base class of anything 2D.

Next a brief description of the class. This text appears in Godot Editor popups for creating Nodes, Resources, and other types.

Description



Description

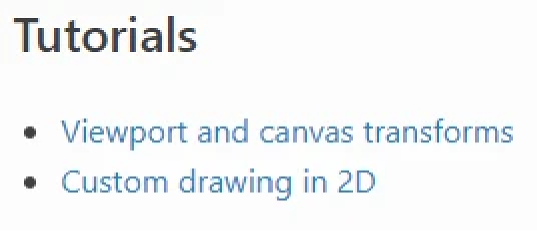
Base class of anything 2D. Canvas items are laid out in a tree; children inherit and extend their parent's transform. `CanvasItem` is extended by Control for anything GUI-related, and by Node2D for anything related to the 2D engine.

Next a more detailed description the class, its features, and its use case(s).

Things you may find here:

1. Specifics of how the class works.
2. Code samples of common use cases.
3. Usage details which are shared between each of the class's methods.
4. Warnings about required dependencies or configuration.
5. Links to other related parts of the Godot API.

Tutorials

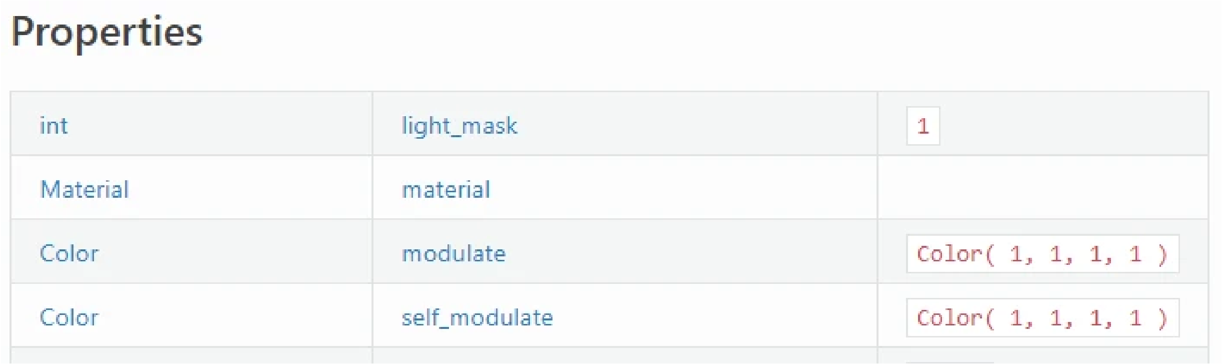


Tutorials

- Viewport and canvas transforms
- Custom drawing in 2D

The page then provides links to parts of the manual which mention or make use of the current class.

Properties



Properties

int	light_mask	`1`
Material	material	
Color	modulate	`Color( 1, 1, 1, 1 )`
Color	self_modulate	`Color( 1, 1, 1, 1 )`

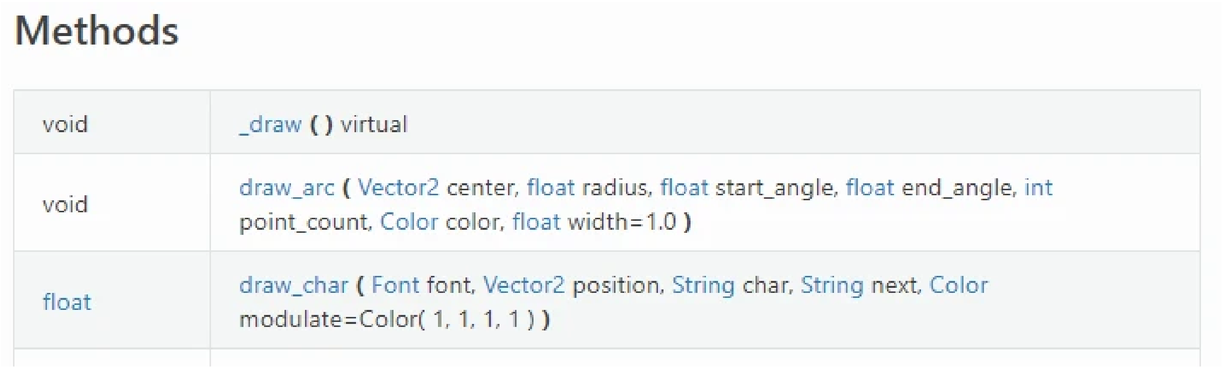
The Properties table lists the variables which belong to each instance of the class, also known as the "properties."

The left column contains the data type of the property. The text is also a link to that data type's Godot API page.

The center column contains the name of the property. The text is also a link to that property's full description on the page. Use this name to get the property's data or set a new value to it.

The right column contains the default value of the property. To initialize it with a different value, you must set a different value via script or the Inspector.

Methods



Methods

void	_draw **( )** virtual
void	draw_arc **(** Vector2 center, float radius, float start_angle, float end_angle, int point_count, Color color, float width=1.0 **)**
float	draw_char **(** Font font, Vector2 position, String char, String next, Color modulate=Color(1, 1, 1, 1) **)**

The Methods table lists the functions which belong to each instance of the class, also known as the "methods."

The left column contains the data type of the method's return value.

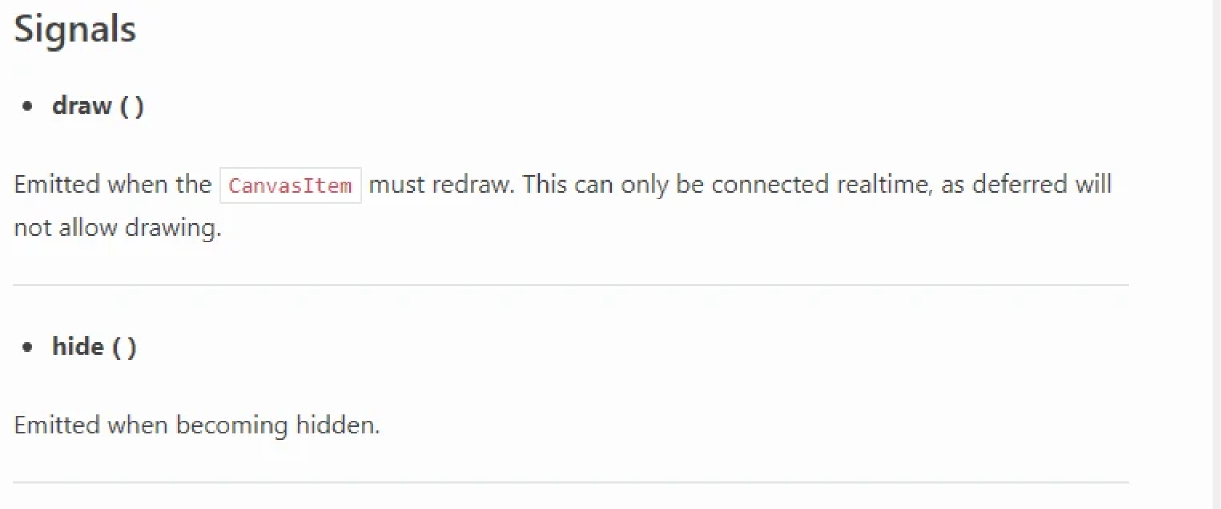
The right column contains the name, parameters, and qualifiers of the method. The name is the text before the opening parenthesis. It is also a link to the method's full description on the page. Use this name to call the method.

For each parameter, the page details its data type, name, and default value, if any.

Possible qualifiers include...

- `const`: the method does not change any data in the class instance.
- `virtual`: the method does nothing but wait for a script to override it.
- `vararg`: the method can accept an arbitrary number of arguments.

Signals



Signals

- **draw ( )**

Emitted when the `CanvasItem` must redraw. This can only be connected realtime, as deferred will not allow drawing.

- **hide ( )**

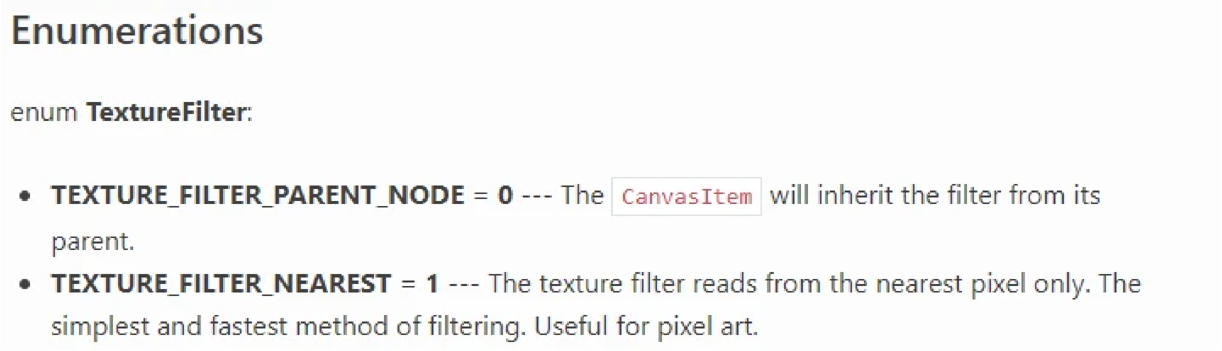
Emitted when becoming hidden.

The Signals list details the names and parameters of events which "signal" a change in game state to other class instances.

Like the Methods table, any parameters will include their data type and name.

Each signal also has a detailed explanation of when the signal is emitted.

Enumerations



Enumerations

enum **TextureFilter**:

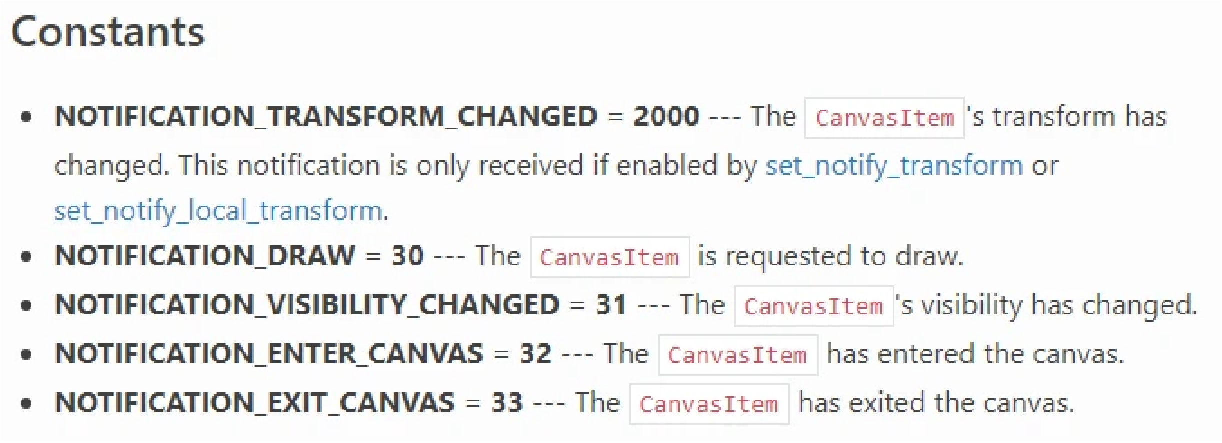
- **TEXTURE_FILTER_PARENT_NODE** = **0** --- The `CanvasItem` will inherit the filter from its parent.
- **TEXTURE_FILTER_NEAREST** = **1** --- The texture filter reads from the nearest pixel only. The simplest and fastest method of filtering. Useful for pixel art.

The Enumerations list details the enumerable data types associated with the current class.

For each enumeration, the page states its name and then lists its possible values.

For each enumeration value, the page states its name, its integer value, and an explanation of its use case(s) and/or affects.

Constants



Constants

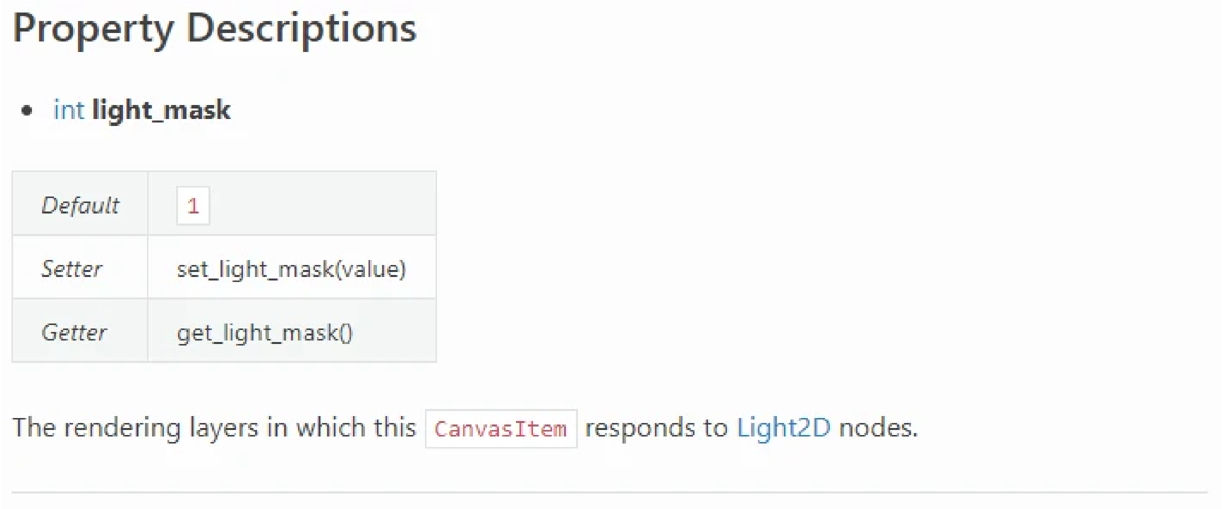
- **NOTIFICATION_TRANSFORM_CHANGED** = **2000** --- The `CanvasItem`'s transform has changed. This notification is only received if enabled by set_notify_transform or set_notify_local_transform.
- **NOTIFICATION_DRAW** = **30** --- The `CanvasItem` is requested to draw.
- **NOTIFICATION_VISIBILITY_CHANGED** = **31** --- The `CanvasItem`'s visibility has changed.
- **NOTIFICATION_ENTER_CANVAS** = **32** --- The `CanvasItem` has entered the canvas.
- **NOTIFICATION_EXIT_CANVAS** = **33** --- The `CanvasItem` has exited the canvas.

The Constants list details named integer constants in the current class.

For each constant, the page states its name, its integer value, and an explanation of its use case(s) and/or affects.

`NOTIFICATION_*` constants' descriptions will state which engine event triggers the notification.

Property Descriptions



Property Descriptions

- int **light_mask**

Default	`1`
Setter	set_light_mask(value)
Getter	get_light_mask()

The rendering layers in which this `CanvasItem` responds to Light2D nodes.

The Property Descriptions list details everything about each property.

It restates the data type and name of the property.

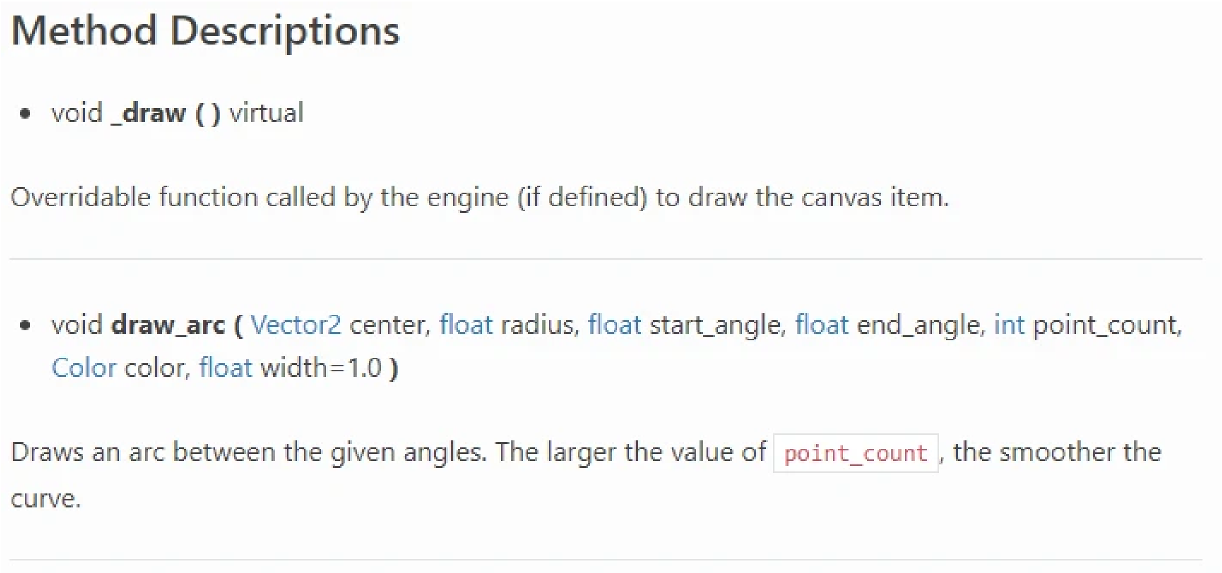
Every property in the Godot API is bound to a pair of setter and getter functions. Using either is equivalent. They are listed here.

Below that is a detailed summary of what the property's data represents, its use case(s) and/or the affects of changing it. It may include code samples and/or links to relevant parts of the Godot API.

Note

Knowing the setter and getter names is useful when one must bind a method name or Callable to something.

Method Descriptions



Method Descriptions

- void **_draw ( )** virtual

Overridable function called by the engine (if defined) to draw the canvas item.

- void **draw_arc (** Vector2 center, float radius, float start_angle, float end_angle, int point_count, Color color, float width=1.0 **)**

Draws an arc between the given angles. The larger the value of `point_count`, the smoother the curve.

The Method Descriptions list details everything about each method.

It restates the method's return data type, parameter names/types/defaults, and qualifiers.

Below that is a detailed summary of what the method does and its use case(s). It may include code samples and/or links to relevant parts of the Godot API.

Debug

- Overview of debugging tools
- Debugger panel
- The Profiler
- Custom performance monitors

Overview of debugging tools

This guide will give you an overview of the available debugging tools in the engine.

Godot comes with a powerful debugger and profilers to track down bugs, inspect your game at runtime, monitor essential metrics, and measure performances. It also offers options to visualize collision boxes and navigation polygons in the running game.

Finally, you have options to debug the game running on a remote device and to reload changes to your scenes or your code while the game is running.

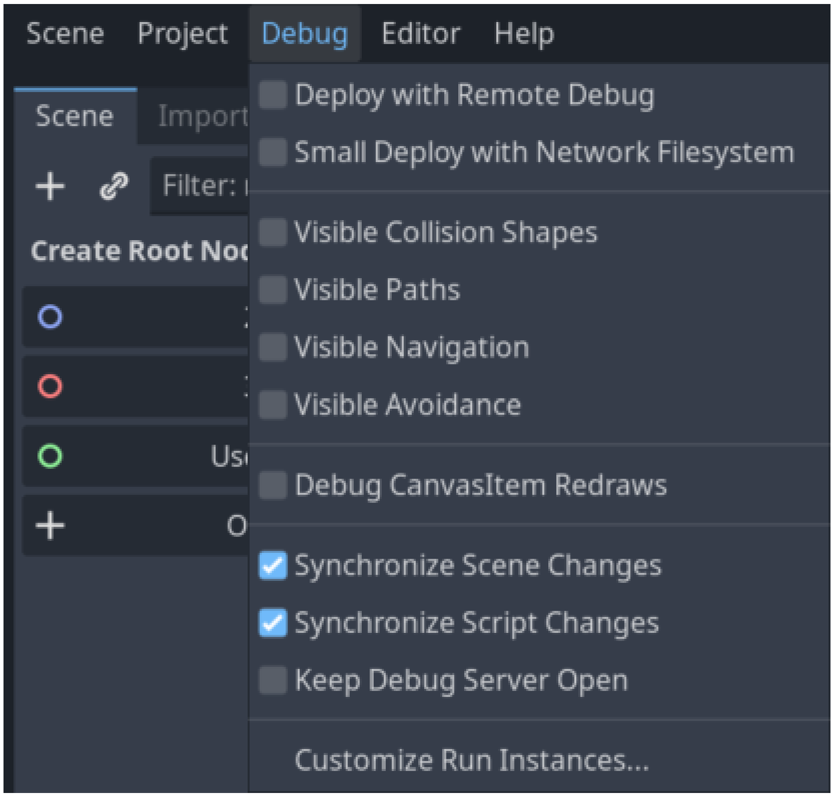
Debugger Panel

Many of Godot's debugging tools are part of the Debugger panel, which you can find information about in Debugger panel.

Debug menu options

There are a few common debug options you can toggle on or off when running your game in the editor, which can help you in debugging your game.

You can find these options in the **Debug** editor menu.


Scene
Project
Debug
Editor
Help
Scene
Import
Filter:
Create Root Noc
Us
O
☐ Deploy with Remote Debug
☐ Small Deploy with Network Filesystem
☐ Visible Collision Shapes
☐ Visible Paths
☐ Visible Navigation
☐ Visible Avoidance
☐ Debug CanvasItem Redraws
☑ Synchronize Scene Changes
☑ Synchronize Script Changes
☐ Keep Debug Server Open
Customize Run Instances...

Here are the descriptions of the options:

Deploy with Remote Debug

When this option is enabled, using one-click deploy will make the executable attempt to connect to this computer's IP so the running project can be debugged. This option is intended to be used for remote debugging (typically with a mobile device). You don't need to enable it to use the GDScript debugger locally.

Small Deploy with Network Filesystem

This option speeds up testing for games with a large footprint on remote devices.

When **Small Deploy with Network Filesystem** is on, instead of exporting the full game, deploying the game builds a minimal executable. The editor then provides files from the project over the network.

Also, on Android, the game is deployed using the USB cable to speed up deployment.

Visible Collision Shapes

When this option is enabled, collision shapes and raycast nodes (for 2D and 3D) will be visible in the running project.

Visible Paths

When this option is enabled, curve resources used by path nodes will be visible in the running project.

Visible Navigation

When this option is enabled, navigation meshes, and polygons will be visible in the running project.

Visible Avoidance

When this option is enabled, avoidance object shapes, radiuses, and velocities will be visible in the running project.

Debug CanvasItem Redraws

When this option is enabled, redraw requests of 2D objects will become visible (as a short flash) in the running project. This is useful to troubleshoot low processor mode.

Synchronize Scene Changes

When this option is enabled, any changes made to the scene in the editor will be replicated in the running project. When used remotely on a device, this is more efficient when the network filesystem option is enabled.

Synchronize Script Changes

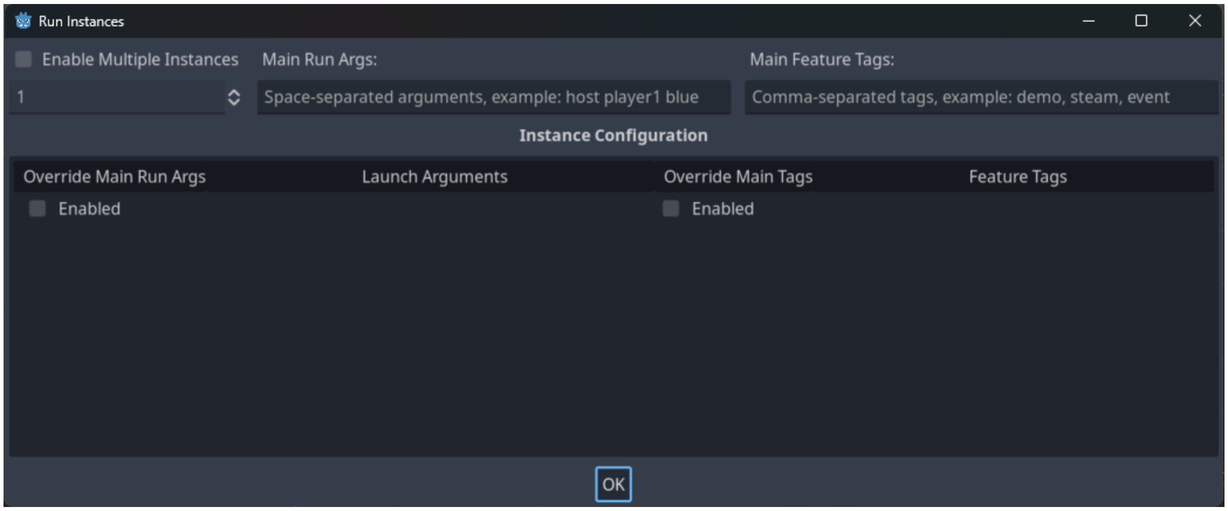
When this option is enabled, any changes made to the script in the editor will be reloaded in the running project. When used remotely on a device, this is more efficient with the network filesystem.

Keep Debug Server Open

When this option is enabled, the editor debug server will stay open and listen for new sessions started outside of the editor itself.

Customize Run Instances...

This opens a dialog allowing you to tell Godot to run multiple instances of the game at once, and to specify the command line arguments for each instance. This is especially useful when building and debugging multiplayer games.


Run Instances
☐ Enable Multiple Instances
Main Run Args:
Main Feature Tags:
1
Space-separated arguments, example: host player1 blue
Comma-separated tags, example: demo, steam, event
Instance Configuration
Override Main Run Args
Launch Arguments
Override Main Tags
Feature Tags
☐ Enabled
☐ Enabled
OK

Enable Multiple Instances

When this option is enabled, the editor will run multiple instances of the project at once when you Run Project.

Below this checkbox is a selector to pick how many instances to run.

Checking the box and setting this to only 1 is the same as not checking this box at all.

Main Run Args

These are the arguments that will be passed to **every** instance of the project when you Run Project, unless you select "Enabled" under "Override Main Run Args" for a specific instance.

Note that these arguments are space-separated.

Tip

These arguments can be accessed in your script by using get_cmdline_args.

Warning

Even if you uncheck "Enable Multiple Instances" these arguments will be passed when you Run Project.

Main Feature Tags

These are the feature tags that will be passed to **every** instance of the project when you Run Project, unless you select "Enabled" under "Override Main Tags" for a specific instance.

Override Main Run Args

When this is enabled, the arguments in the "Main Run Args" field will **not be passed** to this specific instance of the project when you Run Project.

Launch Arguments

These are the arguments that will be passed to this specific instance of the project when you Run Project. They will be **combined with** the "Main Run Args" unless you select "Enabled" under "Override Main Run Args".

Override Main Tags

When this is enabled, the tags in the "Main Feature Tags" field will **not be passed** to this specific instance of the project when you Run Project.

Feature Tags

These are the feature tags that will be passed to this specific instance of the project when you Run Project. They will be **combined with** the "Main Feature Tags" unless you select "Enabled" under "Override Main Tags".

Warning

If you want to pass "User" arguments, that can be accessed with get_cmdline_user_args then you must prefix them with two dashes **and a space** like *-- one two three*.

Be aware that these dashes will apply to arguments added later in the "Launch Arguments" on a per instance basis, which can cause some confusion when combining the *Main Run Args* and *Launch Arguments*.

If you place *-- one two three* in the "Main Run Args" and *-- four five six* in the "Launch Arguments" then the final command line arguments will be *one two three -- four five six*. This is because the *--* is repeated in the "Launch Arguments".

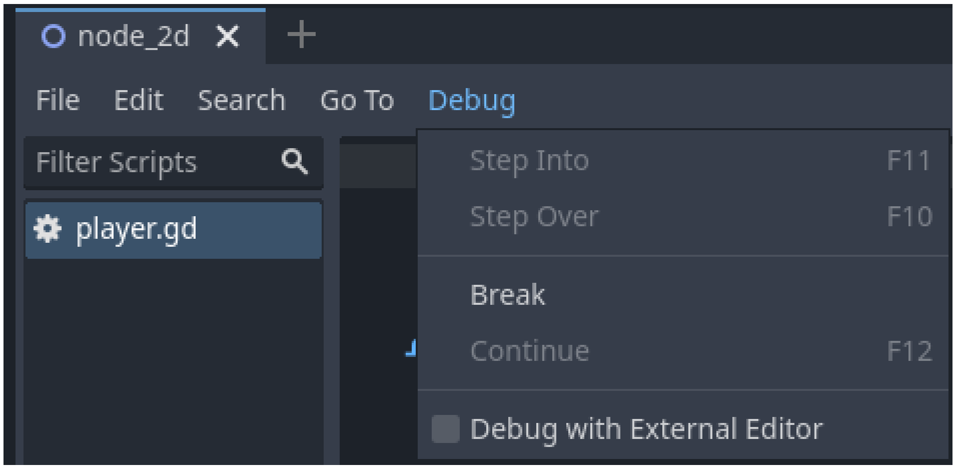
Script editor debug tools and options

The script editor has its own set of debug tools for use with breakpoints and two options. The breakpoint tools can also be found in the **Debugger** tab of the debugger.

Tip

You can create a breakpoint by clicking the gutter in the left of the script editor (on the left of the line numbers). When hovering this gutter, you will see a transparent red dot appearing, which turns into an opaque red dot after the breakpoint is placed by clicking. Click the red dot again to remove the breakpoint. Breakpoints created this way persist across editor restarts, even if the script wasn't saved when exiting the editor.

You can also use the `breakpoint` keyword in GDScript to create a breakpoint that is stored in the script itself. Unlike breakpoints created by clicking in the gutter, this keyword-based breakpoint is persistent across different machines when using version control.


node_2d
File
Edit
Search
Go To
Debug
Filter Scripts
player.gd
Step Into F11
Step Over F10
Break
Continue F12
Debug with External Editor

The **Break** button causes a break in the script like a breakpoint would. **Continue** makes the game continue after pausing at a breakpoint. **Step Over** goes to the next line of code, and **Step Into** goes into a function if possible. Otherwise, it does the same thing as **Step Over**.

The **Debug with External Editor** option lets you debug your game with an external editor. This option is also accessible in **Editor Settings > Debugger**.

When the debugger breaks on a breakpoint, a green triangle arrow is visible in the script editor's gutter. This arrow indicates the line of code the debugger broke on.

Debug project settings

In the project settings, there is a **Debug** category with three subcategories which control different things.

Settings

These are some general settings such as printing the current FPS to the **Output** panel, the maximum amount of functions when profiling and others.

GDScript

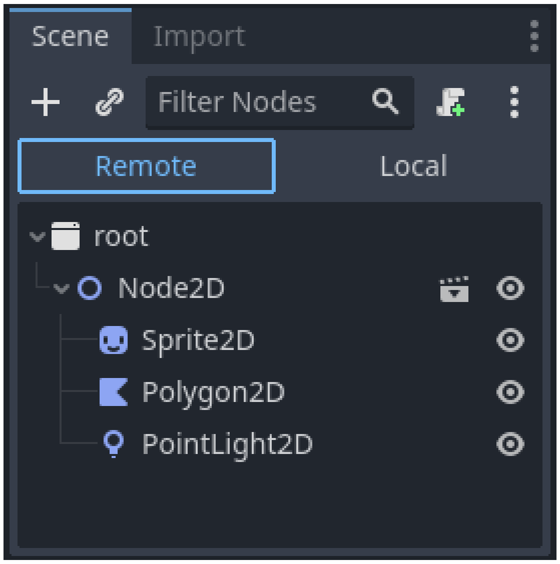
These settings allow you to toggle specific GDScript warnings, such as for unused variables. You can also turn off warnings completely.

Shapes

Shapes are where you can adjust the color of shapes that only appear for debugging purposes, such as collision and navigation shapes.

Remote in scene dock

When running a game in the editor two options appear at the top of the **Scene** dock, **Remote** and **Local**. While using **Remote** you can inspect or change the nodes' parameters in the running project.

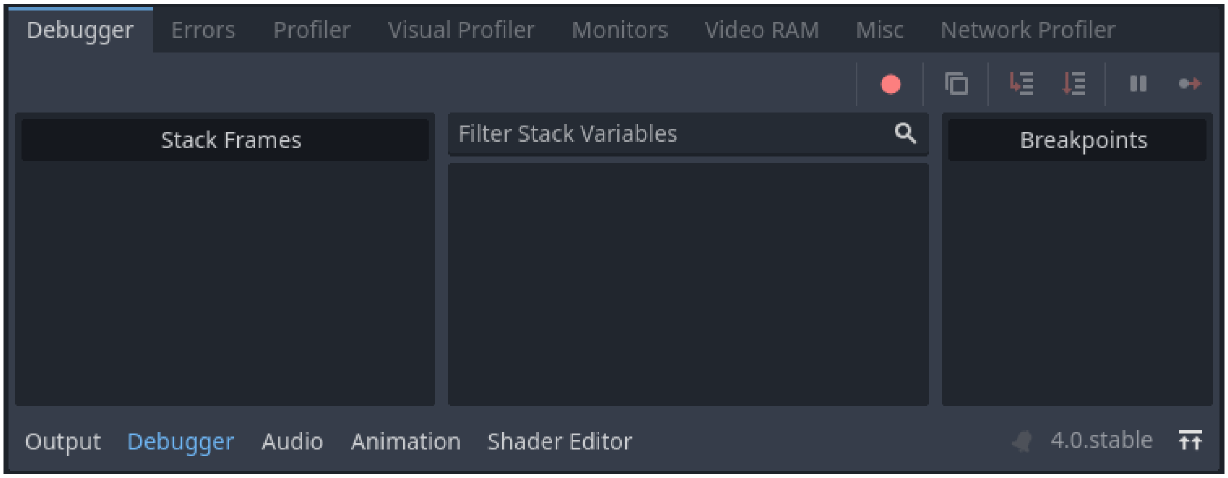

Scene
Import
Filter Nodes
Remote
Local
root
Node2D
Sprite2D
Polygon2D
PointLight2D

Note

Some editor settings related to debugging can be found inside the **Editor Settings**, under the **Network > Debug** and **Debugger** sections.

Debugger panel

Many of Godot's debugging tools, including the debugger, can be found in the debugger panel at the bottom of the screen. Click on **Debugger** to open it.


Debugger
Errors
Profiler
Visual Profiler
Monitors
Video RAM
Misc
Network Profiler
Stack Frames
Filter Stack Variables
Breakpoints
Output
Debugger
Audio
Animation
Shader Editor
4.0.stable

The debugger panel is split into several tabs, each focusing on a specific task.

Debugger

The Debugger tab opens automatically when the GDScript compiler reaches a breakpoint in your code.

It gives you a stack trace [https://en.wikipedia.org/wiki/Stack_trace], information about the state of the object, and buttons to control the program's execution. When the debugger breaks on a breakpoint, a green triangle arrow is visible in the script editor's gutter. This arrow indicates the line of code the debugger broke on.

Tip

You can create a breakpoint by clicking the gutter in the left of the script editor (on the left of the line numbers). When hovering this gutter, you will see a transparent red dot appearing, which turns into an opaque red dot after the breakpoint is placed by clicking. Click the red dot again to remove the breakpoint. Breakpoints created this way persist across editor restarts, even if the script wasn't saved when exiting the editor.

You can also use the `breakpoint` keyword in GDScript to create a breakpoint that is stored in the script itself. Unlike breakpoints created by clicking in the gutter, this keyword-based breakpoint is persistent across different machines when using version control.

You can use the buttons in the top-right corner to:

- Skip all breakpoints. That way, you can save breakpoints for future debugging sessions.
- Copy the current error message.
- **Step Into** the code. This button takes you to the next line of code, and if it's a function, it steps line-by-line through the function.
- **Step Over** the code. This button goes to the next line of code, but it doesn't step line-by-line through functions.
- **Break**. This button pauses the game's execution.
- **Continue**. This button resumes the game after a breakpoint or pause.

Errors

This is where error and warning messages are printed while running the game.

You can disable specific warnings in **Project Settings > Debug > GDScript**.

Profiler

The profiler is used to see what code is running while your project is in use, and how that effects performance.

See also

A detailed explanation of how to use the profiler can be found in the dedicated The Profiler page.

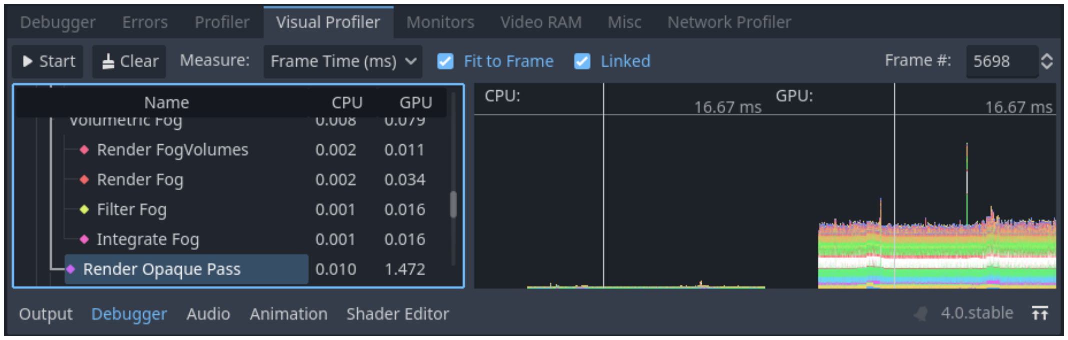
Visual Profiler

The Visual Profiler can be used to monitor what is taking the most time when rendering a frame on the CPU and GPU respectively. This allows tracking sources of potential CPU and GPU bottlenecks caused by rendering.

Warning

The Visual Profiler only measures CPU time taken for rendering tasks, such as performing draw calls. The Visual Profiler does **not** include CPU time taken for other tasks such as scripting and physics. Use the standard Profiler tab to track non-rendering-related CPU tasks.

To use the visual profiler, run the project, switch to the **Visual Profiler** tab within the Debugger bottom panel, then click **Start**:


Debugger
Errors
Profiler
Visual Profiler
Monitors
Video RAM
Misc
Network Profiler
Start
Clear
CPU:
16.67 ms
GPU:
16.67 ms
Output
Debugger
Audio
Animation
Shader Editor
4.0.stable

Measure: Frame Time (ms)
☑ Fit to Frame
☑ Linked
Frame #: 5698

Name	CPU	GPU
Volumetric Fog	0.008	0.079
Render FogVolumes	0.002	0.011
Render Fog	0.002	0.034
Filter Fog	0.001	0.016
Integrate Fog	0.001	0.016
Render Opaque Pass	0.010	1.472

Visual Profiler tab after clicking **Start**, waiting for a few seconds, then clicking **Stop**

You will see categories and results appearing as the profiler is running. Graph lines also appear, with the left side being a CPU framegraph and the right side being a GPU framegraph.

Click **Stop** to finish profiling, which will keep the results visible but frozen in place. Results remain visible after stopping the running project, but not after exiting the editor.

Click on result categories on the left to highlight them in the CPU and GPU graphs on the right. You can also click on the graph to move the cursor to a specific frame number and highlight the selected data type in the result categories on the left.

You can switch the result display between a time value (in milliseconds per frame) or a percentage of the target frametime (which is currently hardcoded to 16.67 milliseconds, or 60 FPS).

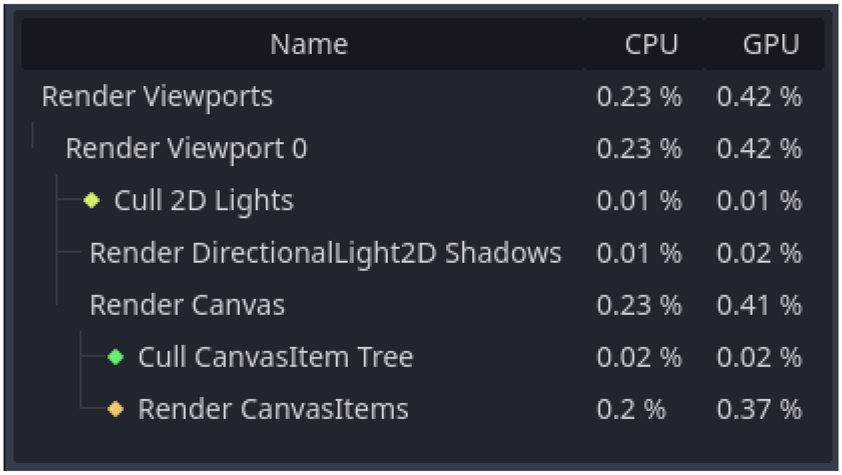
If framerate spikes occur during profiling, this can cause the graph to be poorly scaled. Disable **Fit to Frame** so that the graph will zoom onto the 60 FPS+ portion.

Note

Remember that Visual Profiler results can vary **heavily** based on viewport resolution, which is determined by the window size if using the `disabled` or `canvas_items` stretch modes.

When comparing results across different runs, make sure to use the same viewport size for all runs.

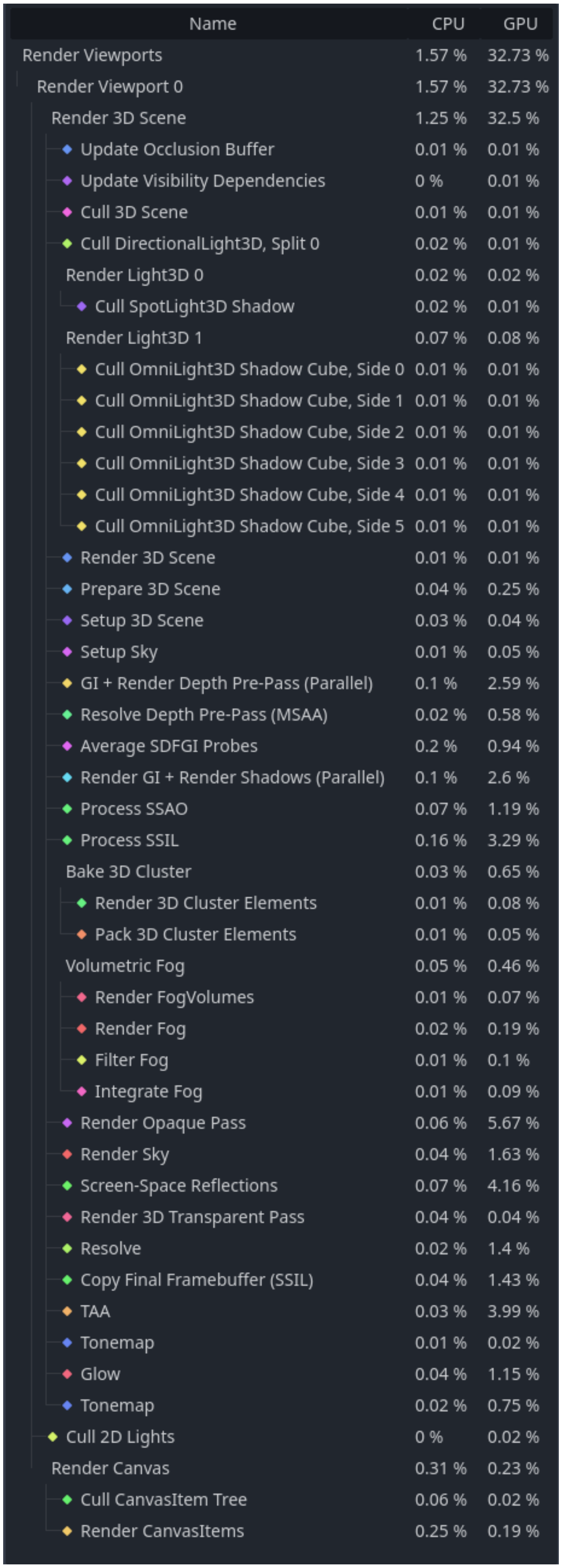
Visual Profiler is supported when using any rendering method (Forward+, Mobile or Compatibility), but the reported categories will vary depending on the current rendering method as well as the enabled graphics features. For example, when using Forward+, a simple 2D scene with shadow-casting lights will result in the following categories appearing:



Name	CPU	GPU
Render Viewports	0.23 %	0.42 %
Render Viewport 0	0.23 %	0.42 %
◆ Cull 2D Lights	0.01 %	0.01 %
Render DirectionalLight2D Shadows	0.01 %	0.02 %
Render Canvas	0.23 %	0.41 %
◆ Cull CanvasItem Tree	0.02 %	0.02 %
◆ Render CanvasItems	0.2 %	0.37 %

Example results from a 2D scene in the Visual Profiler

To give another example with Forward+, a 3D scene with shadow-casting lights and various effects enabled will result in the following categories enabled:



Name	CPU	GPU
Render Viewports	1.57 %	32.73 %
Render Viewport 0	1.57 %	32.73 %
Render 3D Scene	1.25 %	32.5 %
◆ Update Occlusion Buffer	0.01 %	0.01 %
◆ Update Visibility Dependencies	0 %	0.01 %
◆ Cull 3D Scene	0.01 %	0.01 %
◆ Cull DirectionalLight3D, Split 0	0.02 %	0.01 %
Render Light3D 0	0.02 %	0.02 %
◆ Cull SpotLight3D Shadow	0.02 %	0.01 %
Render Light3D 1	0.07 %	0.08 %
◆ Cull OmniLight3D Shadow Cube, Side 0	0.01 %	0.01 %
◆ Cull OmniLight3D Shadow Cube, Side 1	0.01 %	0.01 %
◆ Cull OmniLight3D Shadow Cube, Side 2	0.01 %	0.01 %
◆ Cull OmniLight3D Shadow Cube, Side 3	0.01 %	0.01 %
◆ Cull OmniLight3D Shadow Cube, Side 4	0.01 %	0.01 %
◆ Cull OmniLight3D Shadow Cube, Side 5	0.01 %	0.01 %
◆ Render 3D Scene	0.01 %	0.01 %
◆ Prepare 3D Scene	0.04 %	0.25 %
◆ Setup 3D Scene	0.03 %	0.04 %
◆ Setup Sky	0.01 %	0.05 %
◆ GI + Render Depth Pre-Pass (Parallel)	0.1 %	2.59 %
◆ Resolve Depth Pre-Pass (MSAA)	0.02 %	0.58 %
◆ Average SDFGI Probes	0.2 %	0.94 %
◆ Render GI + Render Shadows (Parallel)	0.1 %	2.6 %
◆ Process SSAO	0.07 %	1.19 %
◆ Process SSIL	0.16 %	3.29 %
Bake 3D Cluster	0.03 %	0.65 %
◆ Render 3D Cluster Elements	0.01 %	0.08 %
◆ Pack 3D Cluster Elements	0.01 %	0.05 %
Volumetric Fog	0.05 %	0.46 %
◆ Render FogVolumes	0.01 %	0.07 %
◆ Render Fog	0.02 %	0.19 %
◆ Filter Fog	0.01 %	0.1 %
◆ Integrate Fog	0.01 %	0.09 %
◆ Render Opaque Pass	0.06 %	5.67 %
◆ Render Sky	0.04 %	1.63 %
◆ Screen-Space Reflections	0.07 %	4.16 %
◆ Render 3D Transparent Pass	0.04 %	0.04 %
◆ Resolve	0.02 %	1.4 %
◆ Copy Final Framebuffer (SSIL)	0.04 %	1.43 %
◆ TAA	0.03 %	3.99 %
◆ Tonemap	0.01 %	0.02 %
◆ Glow	0.04 %	1.15 %
◆ Tonemap	0.02 %	0.75 %
◆ Cull 2D Lights	0 %	0.02 %
Render Canvas	0.31 %	0.23 %
◆ Cull CanvasItem Tree	0.06 %	0.02 %
◆ Render CanvasItems	0.25 %	0.19 %

Example results from a 3D scene in the Visual Profiler

Notice how in the 3D example, several of the categories have **(Parallel)** appended to their name. This hints that multiple tasks are being performed in parallel on the GPU. This generally means that disabling only one of the features involved won't improve performance as much as anticipated, as the other task still needs to be performed sequentially.

Network Profiler

The Network Profiler contains a list of all the nodes that communicate over the multiplayer API and, for each one, some counters on the amount of incoming and outgoing network interactions. It also features a bandwidth meter that displays the total bandwidth usage at any given moment.

Note

The bandwidth meter does **not** take the High-level multiplayer API's own compression system into account. This means that changing the compression algorithm used will not change the metrics reported by the bandwidth meter.

Monitors

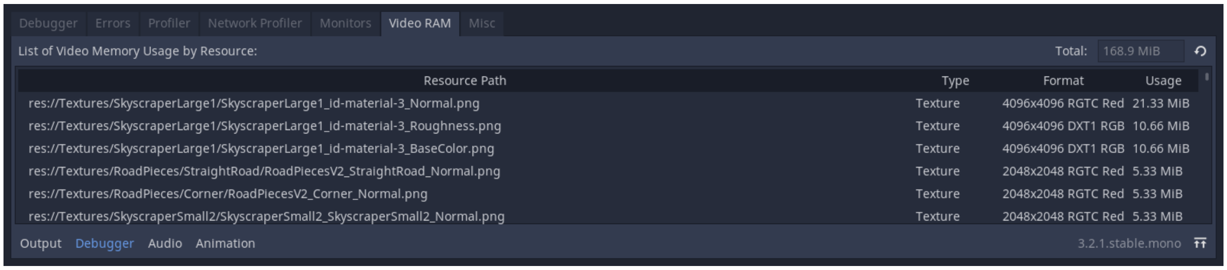
The monitors are graphs of several aspects of the game while its running such as FPS, memory usage, how many nodes are in a scene and more. All monitors keep track of stats automatically, so even if one monitor isn't open while the game is running, you can open it later and see how the values changed.

See also

In addition to the default performance monitors, you can also create custom performance monitors to track arbitrary values in your project.

Video RAM

The **Video RAM** tab shows the video RAM usage of the game while it is running. It provides a list of every resource using video RAM by resource path, the type of resource it is, what format it is in, and how much Video RAM that resource is using. There is also a total video RAM usage number at the top right of the panel.



List of Video Memory Usage by Resource:

Total: 168.9 MiB

Resource Path	Type	Format	Usage
res://Textures/SkyscraperLarge1/SkyscraperLarge1_id-material-3_Normal.png	Texture	4096x4096 RGTC Red	21.33 MiB
res://Textures/SkyscraperLarge1/SkyscraperLarge1_id-material-3_Roughness.png	Texture	4096x4096 DXT1 RGB	10.66 MiB
res://Textures/SkyscraperLarge1/SkyscraperLarge1_id-material-3_BaseColor.png	Texture	4096x4096 DXT1 RGB	10.66 MiB
res://Textures/RoadPieces/StraightRoad/RoadPiecesV2_StraightRoad_Normal.png	Texture	2048x2048 RGTC Red	5.33 MiB
res://Textures/RoadPieces/Corner/RoadPiecesV2_Corner_Normal.png	Texture	2048x2048 RGTC Red	5.33 MiB
res://Textures/SkyscraperSmall2/SkyscraperSmall2_SkyscraperSmall2_Normal.png	Texture	2048x2048 RGTC Red	5.33 MiB

Misc

The **Misc** tab contains tools to identify the control nodes you are clicking at runtime:

- **Clicked Control** tells you where the clicked node is in the scene tree.
- **Clicked Control Type** tells you the type of the node you clicked is.

The Profiler

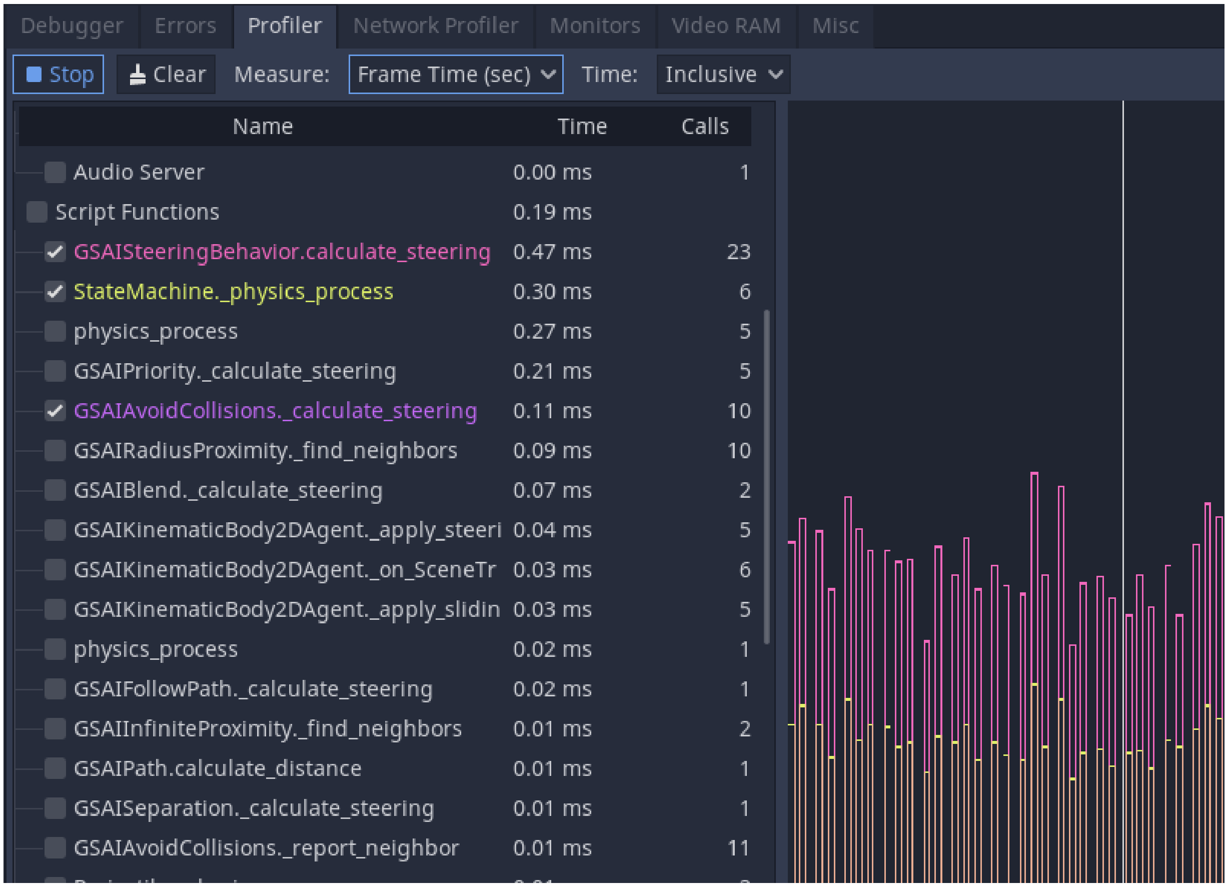
You run your game from Godot and play around. It's fun, it's becoming feature complete, and you feel it's getting close to release.

But then, you open the skill tree, and it grinds to a halt as something snags in your code. Watching the skill tree scroll by like it's a slide show is unacceptable. What went wrong? Is it positioning the skill tree elements, the UI, or rendering?

You could try to optimize everything and run the game repeatedly, but you can be smarter about this and narrow down the possibilities. Enter Godot's profiler.

An overview of the profiler

You can open the profiler by opening the **Debugger** panel and clicking on the **Profiler** tab.



Measure: Frame Time (sec)
Time: Inclusive

Name	Time	Calls
☐ Audio Server	0.00 ms	1
☐ Script Functions	0.19 ms	
☑ GSAISteeringBehavior.calculate_steering	0.47 ms	23
☑ StateMachine._physics_process	0.30 ms	6
☐ physics_process	0.27 ms	5
☐ GSAIPriority._calculate_steering	0.21 ms	5
☑ GSAIAvoidCollisions._calculate_steering	0.11 ms	10
☐ GSAIRadiusProximity._find_neighbors	0.09 ms	10
☐ GSAIBlend._calculate_steering	0.07 ms	2
☐ GSAIKinematicBody2DAgent._apply_steeri	0.04 ms	5
☐ GSAIKinematicBody2DAgent._on_SceneTr	0.03 ms	6
☐ GSAIKinematicBody2DAgent._apply_slidin	0.03 ms	5
☐ physics_process	0.02 ms	1
☐ GSAIFollowPath._calculate_steering	0.02 ms	1
☐ GSAIInfiniteProximity._find_neighbors	0.01 ms	2
☐ GSAIPath.calculate_distance	0.01 ms	1
☐ GSAISeparation._calculate_steering	0.01 ms	1
☐ GSAIAvoidCollisions._report_neighbor	0.01 ms	11

Godot's profiler does not automatically run because profiling is performance-intensive. It has to continually measure everything happening in the game and report back to the debugger, so it's off by default.

To begin profiling, run your game then focus back on the editor. Click on the **Start** button in the top-left corner of the **Profiler** tab.

Note

The profiler does not currently support C# scripts. C# scripts can be profiled using JetBrains Rider and JetBrains dotTrace with the Godot support plugin.

You can clear the data by clicking the **Clear** button anytime. Use the **Measure** drop-down menu to change the type of data you measure. The measurements panel and the graph will update accordingly.

The measured data

The profiler's interface is split into two. There is a list of functions on the left and the performance graph on the right.

The main measurements are frame time, physics frame, idle time, and physics time.

- The **frame time** is the time it takes Godot to execute all the logic for an entire image, from physics to rendering.
- **Physics frame** is the time Godot has allocated between physics updates. In an ideal scenario, the frame time is whatever you chose: 16.66 milliseconds by default, which corresponds to 60FPS. It's a frame of reference you can use for everything else around it.
- **Idle time** is the time Godot took to update logic other than physics, such as code that lives in *_process* or timers and cameras set to update on **Idle**.
- **Physics time** is the time Godot took to update physics tasks, like *_physics_process* and built-in nodes set to **Physics** update.

Note

Frame Time includes rendering time. Say you find a mysterious spike of lag in your game, but your physics and scripts are all running fast. The delay could be due to the appearance of particles or visual effects!

By default, Godot ticks on Frame Time and Physics Time. This gives you an overview of how long each frame takes relative to the allocated desired physics FPS. You can toggle functions on and off by clicking the checkboxes on the left. Other facilities make appearances as you go down the list, like Physics 2D, Physics, and Audio, before reaching Script functions, where your code appears.

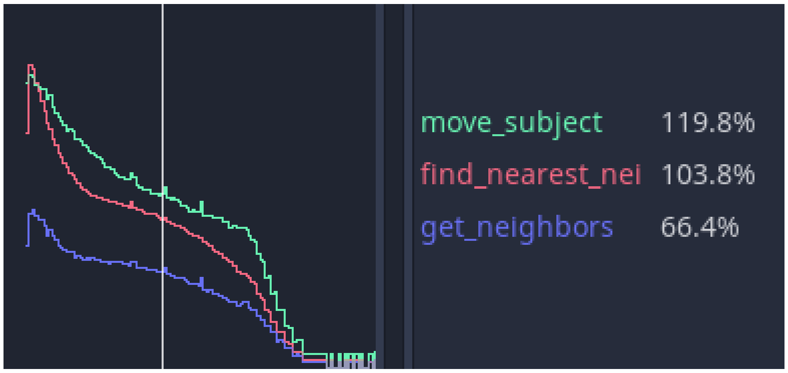
If you click on the graph, you change which frame's information appears on the left. In the top right, there is also a frame counter where you can manually adjust the frame you are looking at more granularly.

Scope of measurement and measurement windows

You can change what measurement you are looking at using the **Measure** drop-down menu. By default, it starts with Frame Time and lists the time it takes to go through the frame in milliseconds. The average time is the average time any given function took when called more than once. For example, a function that took 0.05 milliseconds to run five times should give you an average of 0.01 milliseconds.

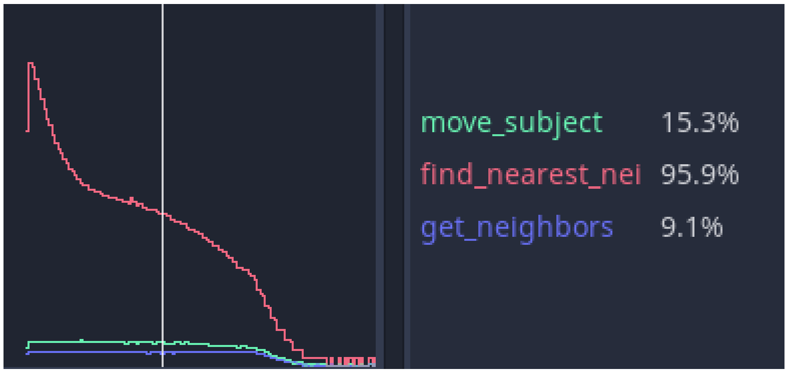
If accurate milliseconds count is not important, and you want to see proportions of time relative to the rest of the frame, use percentage measurements. Frame % is relative to Frame Time, and Physics % is relative to Physics Time.

The last option is the scope of the time. **Inclusive** measures the time a function took **with** any nested function calls. For example:



get_neighbors, *find_nearest_neighbor* and *move_subject* all took a lot of time. You could be fooled into thinking that this is because all three of them are slow.

But when changed to **Self**, Godot measures the time spent in the function body without considering function calls it made itself.



You can see that *get_neighbors* and *move_subject* have lost a lot of their importance. In effect, that means that *get_neighbors* and *move_subject* have spent more time waiting for some other function call to finish than not, and *find_nearest_neighbor* is **actually** slow.

Debugging slow code with the profiler

Finding slow code with the profiler boils down to running your game and watching the performance graph as it draws. When an unacceptable spike occurs in the frame time, you can click on the graph to pause your game and narrow the _Frame #_ to the spike's start. You may need to jump back and forth between frames and functions to find the root cause.

Under the Script functions, turn on the checkboxes for some functions to find which take time. These are the functions you need to review and optimize.

Measuring manually in microseconds

If your function is complex, it could be challenging to figure out which part needs optimization. Is it your math or the way you access other pieces of data to do the math with? Is it the *for* loop? The *if* statements?

You can narrow down the measurement by manually counting ticks as the code runs with some temporary functions. The two functions are part of the *Time* class object. They are *get_ticks_msec* and *get_ticks_usec*. The first measures in milliseconds (1,000 per second), and the second measures in microseconds (1,000,000 per second).

Either one returns the amount of time since the game engine started in their respective time frame.

If you wrap a piece of code with a start and end count of microseconds, the difference between the two is the amount of time it took to run that piece of code.

GDScript

```
# Measuring the time it takes for worker_function() to run
var start = Time.get_ticks_usec()
```

```
worker_function()
var end = Time.get_ticks_usec()
var worker_time = (end-start)/1000000.0

# Measuring the time spent running a calculation over each
element of an array
start = Time.get_ticks_usec()
for calc in calculations:
    result = pow(2, calc.power) * calc.product
end = Time.get_ticks_usec()
var loop_time = (end-start)/1000000.0

print("Worker time: %s\nLoop time: %s" % [worker_time,
loop_time])
```

As you become a more experienced programmer, this technique becomes less necessary. You begin to learn what parts of a running program are slow. Knowing that loops and branches can be slow comes from experience, and you gain experience by measuring and doing research.

But between the profiler and the ticks functions, you should have enough to get started finding which parts of your code need optimization.

Custom performance monitors

Introduction

As explained in the Debugger panel documentation, Godot features a **Debugger > Monitors** bottom panel that allows tracking various values with graphs showing their evolution over time. The data for those graphs is sourced from the engine's Performance singleton.

Since Godot 4.0, you can declare custom values to be displayed in the Monitors tab. Example use cases for custom performance monitors include:

- Displaying performance metrics that are specific to your project. For instance, in a voxel game, you could create a performance monitor to track the number of chunks that are loaded every second.
- Displaying in-game metrics that are not strictly related to performance, but are still useful to graph for debugging purposes. For instance, you could track the number of enemies present in the game to make sure your spawning mechanic works as intended.

Creating a custom performance monitor

In this example, we'll create a custom performance monitor to track how many enemies are present in the currently running project.

The main scene features a Timer node with the following script attached:

```
extends Timer


func _ready():
    # The slash delimiter is used to determine the category of the monitor.
    # If there is no slash in the monitor name, a generic "Custom" category
    # will be used instead.
    Performance.add_custom_monitor("game/enemies", get_enemy_count)
    timeout.connect(_on_timeout)
    # Spawn 20 enemies per second.
    wait_time = 0.05
    start()


func _on_timeout():
    var enemy = preload("res://enemy.tscn").instantiate()
    get_parent().add_child(enemy)


# This function is called every time the performance monitor is queried
# (this occurs once per second in the editor, more if called manually).
# The function must return a number greater than or equal to 0 (int or float).
func get_enemy_count():
    return get_tree().get_nodes_in_group("enemies").size()
```

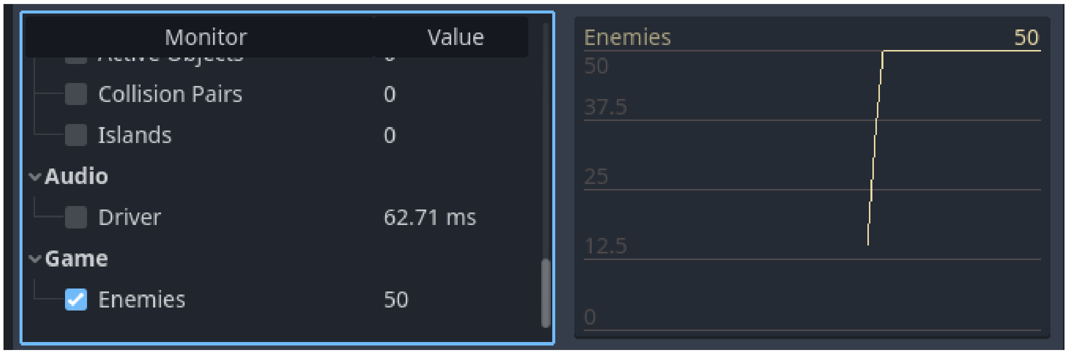
The second parameter of Performance.add_custom_monitor is a Callable.

`enemy.tscn` is a scene with a Node2D root node and Timer child node. The Node2D has the following script attached:

```
extends Node2D


func _ready():
    add_to_group("enemies")
    $Timer.timeout.connect(_on_timer_timeout)
    # Despawn enemies 2.5 seconds after they spawn.
    $Timer.wait_time = 2.5
    $Timer.start()


func _on_timer_timeout():
    queue_free()
```

In this example, since we spawn 20 enemies per second, and each enemy despawns 2.5 seconds after they spawn, we expect the number of enemies present in the scene to stabilize to 50. We can make sure about this by looking at the graph.

To visualize the graph created from this custom performance monitor, run the project, switch to the editor while the project is running and open **Debugger > Monitors** at the bottom of the editor window. Scroll down to the newly available **Game** section and check **Enemies**. You should see a graph appearing as follows:



Monitor	Value
☐ Collision Pairs	0
☐ Islands	0
Audio	
☐ Driver	62.71 ms
Game	
☑ Enemies	50

Example editor graph from a custom performance monitor

Note

The performance monitor handling code doesn't have to live in the same script as the nodes themselves. You may choose to move the performance monitor registration and getter function to an autoload instead.

Querying a performance monitor in a project

If you wish to display the value of the performance monitor in the running project's window (rather than the editor), use `Performance.get_custom_monitor("category/name")` to fetch the value of the custom monitor. You can display the value using a Label, RichTextLabel, Custom drawing in 2D, 3D text, etc.

This method can be used in exported projects as well (debug and release mode), which allows you to create visualizations outside the editor.

Idle and Physics Processing

Games run in a loop. Each frame, you need to update the state of your game world before drawing it on screen. Godot provides two virtual methods in the Node class to do so: Node._process() and Node._physics_process(). If you define either or both in a script, the engine will call them automatically.

There are two types of processing available to you:

1. **Idle processing** allows you to run code that updates a node every frame, as often as possible.
2. **Physics processing** happens at a fixed rate, 60 times per second by default. This is independent of your game's actual framerate, and keeps physics running smoothly. You should use it for anything that involves the physics engine, like moving a body that collides with the environment.

You can activate idle processing by defining the `_process()` method in a script. You can turn it off and back on by calling Node.set_process().

The engine calls this method every time it draws a frame:

GDScript

```
func _process(delta):
    # Do something...
    pass
```

C#

```
public override void _Process(double delta)
{
    // Do something...
}
```

Keep in mind that the frequency at which the engine calls `_process()` depends on your application's framerate, which varies over time and across devices.

The function's `delta` parameter is the time elapsed in seconds since the previous call to `_process()`. Use this parameter to make calculations independent of the framerate. For example, you should always multiply a speed value by `delta` to animate a moving object.

Physics processing works with a similar virtual function: `_physics_process()`. Use it for calculations that must happen before each physics step, like moving a character that collides with the game world. As mentioned above, `_physics_process()` runs at fixed time intervals as much as possible to keep the physics interactions stable. You can change the interval between physics steps in the Project Settings, under Physics -> Common -> Physics Fps. By default, it's set to run 60 times per second.

The engine calls this method before every physics step:

GDScript

```
func _physics_process(delta):
    # Do something...
    pass
```

C#

```
public override void _PhysicsProcess(double delta)
{
    // Do something...
}
```

The function `_process()` is not synchronized with physics. Its rate depends on hardware and game optimization. It also runs after the physics step in single-threaded games.

You can see the `_process()` function at work by creating a scene with a single Label node, with the following script attached to it:

GDScript

```
extends Label

var time = 0

func _process(delta):
    time += delta
    text = str(time) # 'text' is a built-in Label property.
```

C#

```
using Godot;

public partial class CustomLabel : Label
{
    private double _time;

    public override void _Process(double delta)
    {
        _time += delta;
        Text = _time.ToString(); // 'Text' is a built-in Label
property.
    }
}
```

When you run the scene, you should see a counter increasing each frame.

Groups

Groups in Godot work like tags in other software. You can add a node to as many groups as you want. Then, in code, you can use the SceneTree to:

- Get a list of nodes in a group.
- Call a method on all nodes in a group.
- Send a notification to all nodes in a group.

This is a useful feature to organize large scenes and decouple code.

Managing groups

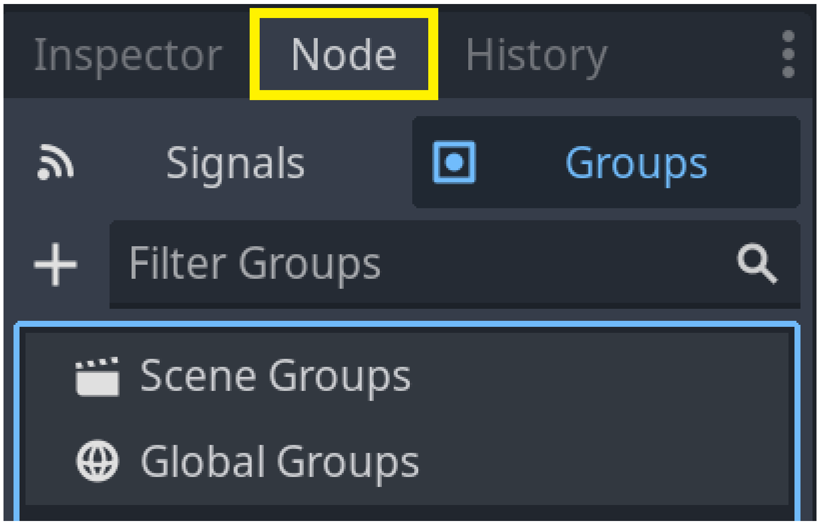
Groups are created by adding a node to a new group name, and likewise they are removed by removing all nodes from a given group.

There are two ways to add/remove nodes to groups:

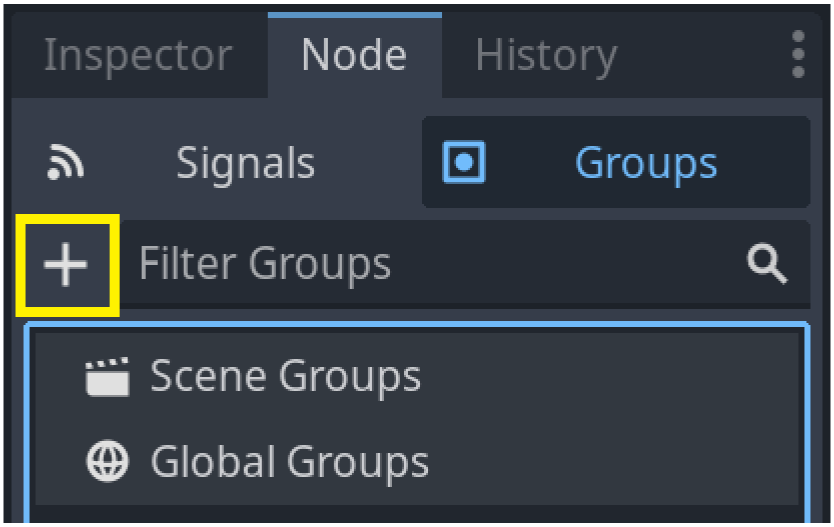
- During design, by using the Node dock in the editor, or the Global Groups in project settings.
- During execution, by calling Node.add_to_group() or Node.remove_from_group().

Using the Node dock

You can create new groups using the Groups tab in the Node dock.


Inspector
Node
History
Signals
Groups
Filter Groups
Scene Groups
Global Groups

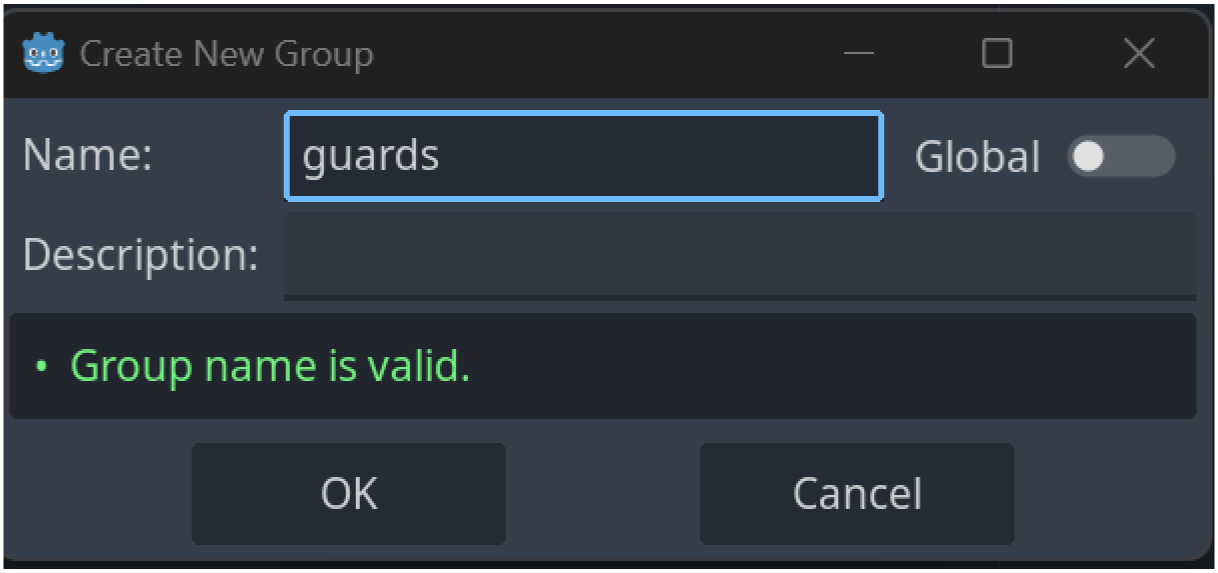
Select one or more nodes in the Scene dock then click the add button with the + symbol.


Inspector
Node
History
Signals
Groups
Filter Groups
Scene Groups
Global Groups

You should now see the Create New Group modal appear. Write the group name in the field.

You can optionally mark the option "Global", which will make the group visible project-wide, and able to be reused in any project scene. This will also allow you to give it a description.

When done, press Ok to create it.


Create New Group
Global
OK
Cancel

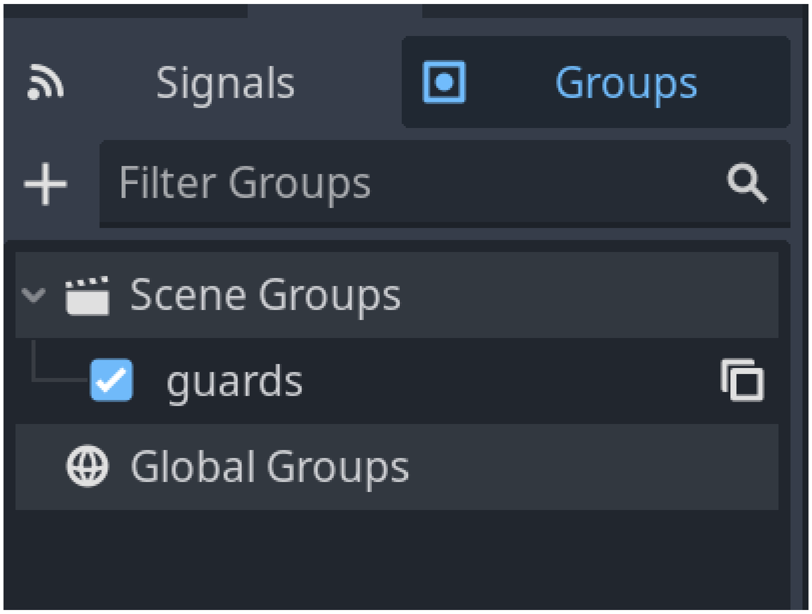
Name: guards

Description:

- Group name is valid.

You should see the new groups appear in the Groups tab under Scene Groups if the Global option was unmarked, or under Global Groups if that option was marked.

Selected Node(s) from the Scene dock can be added into groups by marking the checkbox on the left side of the groups in the Groups dock. The node(s) you had selected when creating a new group will be automatically checked.


Signals
Groups
Filter Groups
Scene Groups
Global Groups

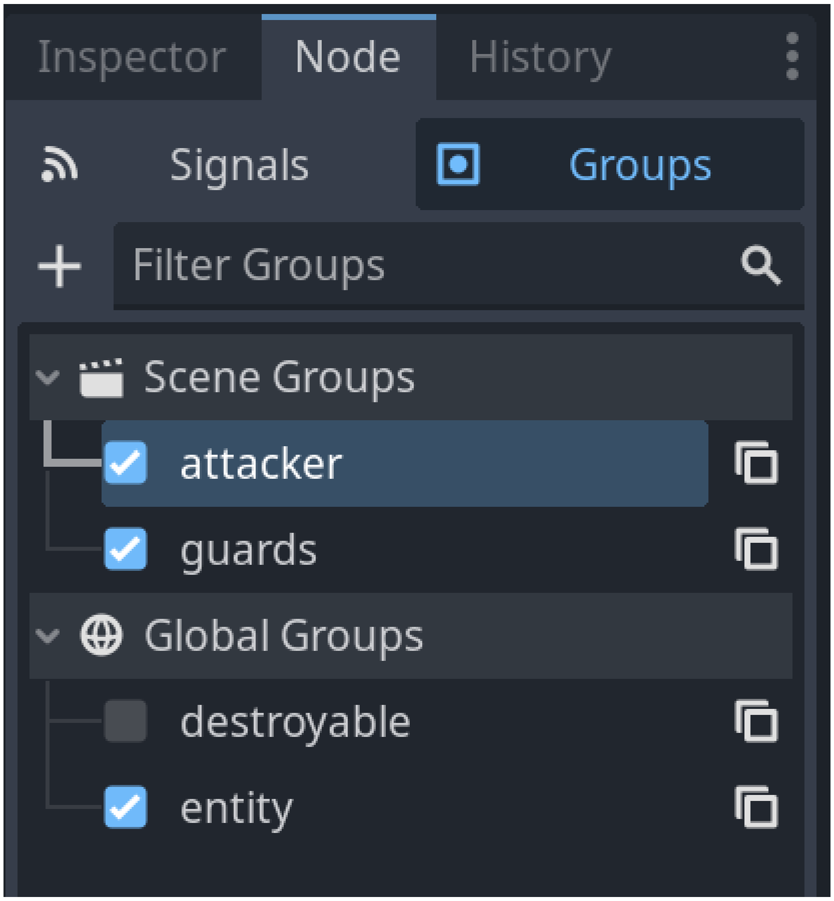
☑ guards

All groups present in the project that were marked as Global, created from any scene, will be visible under Global Groups.

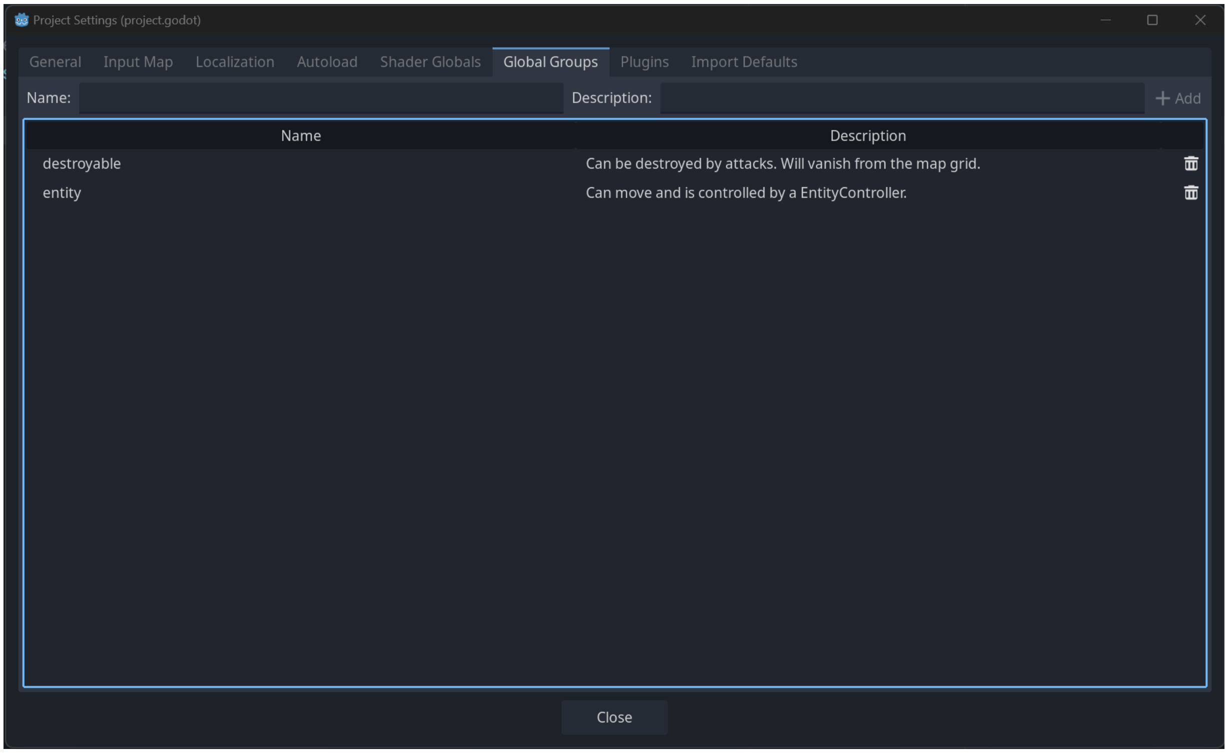
Any other group derived from nodes in the current scene will appear under Scene Groups.

Warning

The same underlying logic is used for both Global and Scene groups. Groups with the same name are considered one and the same. This feature is purely organizational.


Inspector
Node
History
Signals
Groups
Filter Groups
Scene Groups
attacker
guards
Global Groups
destroyable
entity

You can manage Global Groups in the Global Groups dock, inside Project Settings. There, you will be able to add new global groups, or change existing groups' names and descriptions.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Import Defaults
Name:
Description:
Add
Close

Name	Description
destroyable	Can be destroyed by attacks. Will vanish from the map grid.
entity	Can move and is controlled by a EntityController.

Using code

You can also manage groups from scripts. The following code adds the node to which you attach the script to the `guards` group as soon as it enters the scene tree.

GDScript

```
func _ready():
    add_to_group("guards")
```

C#

```
public override void _Ready()
{
    base._Ready();

    AddToGroup("guards");
}
```

Imagine you're creating an infiltration game. When an enemy spots the player, you want all guards and robots to be on alert.

In the fictional example below, we use `SceneTree.call_group()` to alert all enemies that the player was spotted.

GDScript

```
func _on_player_spotted():
    get_tree().call_group("guards", "enter_alert_mode")
```

C#

```
public void _OnPlayerDiscovered()
{
    GetTree().CallGroup("guards", "enter_alert_mode");
}
```

The above code calls the function `enter_alert_mode` on every member of the group `guards`.

To get the full list of nodes in the `guards` group as an array, you can call SceneTree.get_nodes_in_group():

GDScript

```
var guards = get_tree().get_nodes_in_group("guards")
```

C#

```
var guards = GetTree().GetNodesInGroup("guards");
```

The SceneTree class provides many more useful methods to interact with scenes, their node hierarchy, and groups. It allows you to switch scenes easily or reload them, quit the game or pause and unpause it. It also provides useful signals.

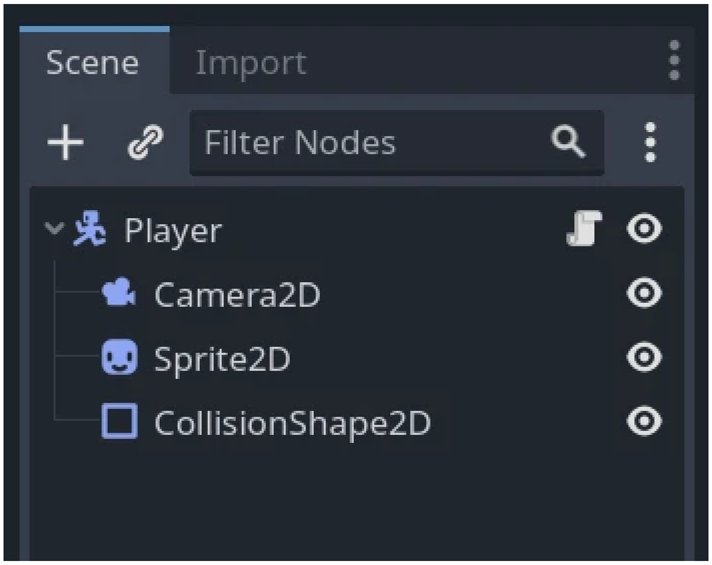
Nodes and scene instances

This guide explains how to get nodes, create nodes, add them as a child, and instantiate scenes from code.

Getting nodes

You can get a reference to a node by calling the Node.get_node() method. For this to work, the child node must be present in the scene tree. Getting it in the parent node's `_ready()` function guarantees that.

If, for example, you have a scene tree like this, and you want to get a reference to the Sprite2D and Camera2D nodes to access them in your script.


Scene
Import
Filter Nodes
Player
Camera2D
Sprite2D
CollisionShape2D

To do so, you can use the following code.

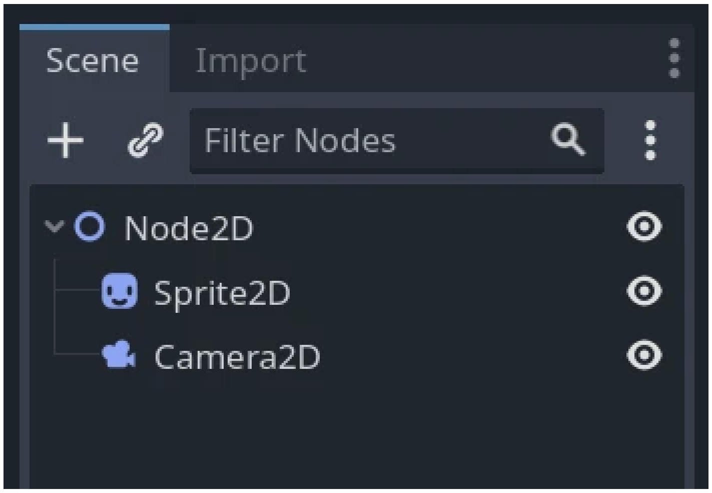
GDScript

```
var sprite2d
var camera2d

func _ready():
    sprite2d = get_node("Sprite2D")
    camera2d = get_node("Camera2D")
```

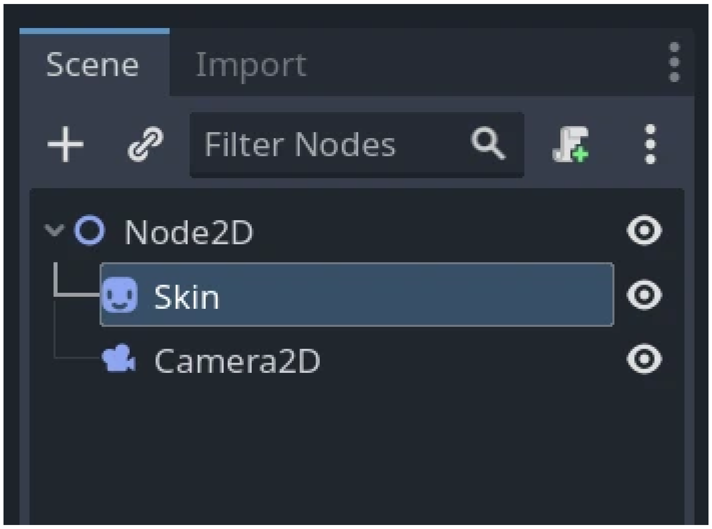
C#

```
private Sprite2D _sprite2D;
private Camera2D _camera2D;

public override void _Ready()
{
    base._Ready();

    _sprite2D = GetNode<Sprite2D>("Sprite2D");
    _camera2D = GetNode<Camera2D>("Camera2D");
}
```

Note that you get nodes using their name, not their type. Above, "Sprite2D" and "Camera2D" are the nodes' names in the scene.



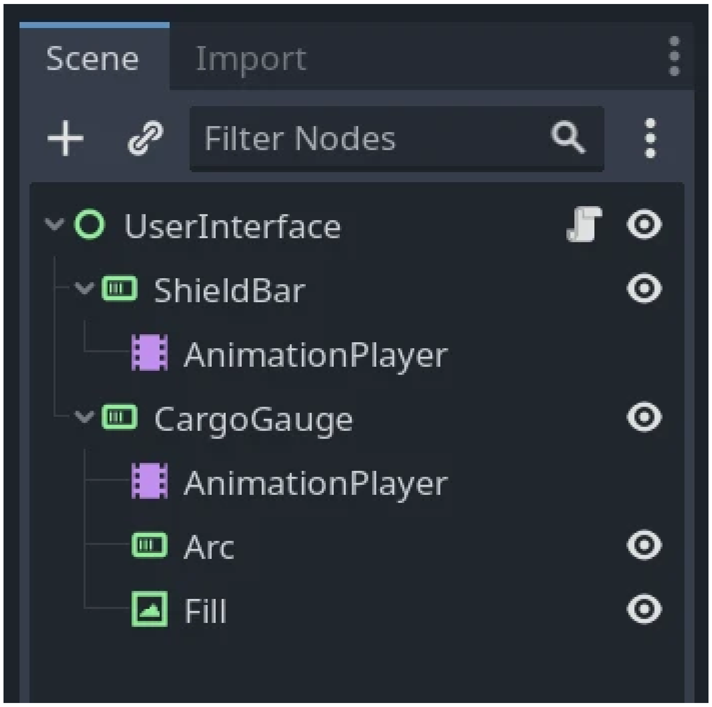
If you rename the Sprite2D node as Skin in the Scene dock, you have to change the line that gets the node to `get_node("Skin")` in the script.


Scene
Import
Filter Nodes
Node2D
Skin
Camera2D

Node paths

When getting a reference to a node, you're not limited to getting a direct child. The `get_node()` function supports paths, a bit like when working with a file browser. Add a slash to separate nodes.

Take the following example scene, with the script attached to the UserInterface node.


Scene
Import
Filter Nodes
UserInterface
ShieldBar
AnimationPlayer
CargoGauge
AnimationPlayer
Arc
Fill

To get the AnimationPlayer node, you would use the following code.

GDScript

```
var animation_player

func _ready():
    animation_player = get_node("ShieldBar/AnimationPlayer")
```

C#

```
private AnimationPlayer _animationPlayer;

public override void _Ready()
{
    base._Ready();

    _animationPlayer = GetNode<AnimationPlayer>
("ShieldBar/AnimationPlayer");
}
```

Note

As with file paths, you can use ".." to get a parent node. The best practice is to avoid doing that though not to break encapsulation. You can also start the path with a forward slash to make it absolute, in which case your topmost node would be "/root", the application's predefined root viewport.

Syntactic sugar

You can use two shorthands to shorten your code in GDScript. Firstly, putting the `@onready` annotation before a member variable makes it initialize right before the `_ready()` callback.

```
@onready var sprite2d = get_node("Sprite2D")
```

There is also a short notation for `get_node()`: the dollar sign, "$". You place it before the name or path of the node you want to get.

```
@onready var sprite2d = $Sprite2D
@onready var animation_player = $ShieldBar/AnimationPlayer
```

Creating nodes

To create a node from code, call its `new()` method like for any other class-based datatype.

You can store the newly created node's reference in a variable and call `add_child()` to add it as a child of the node to which you attached the script.

GDScript

```
var sprite2d

func _ready():
    var sprite2d = Sprite2D.new() # Create a new Sprite2D.
    add_child(sprite2d) # Add it as a child of this node.
```

C#

```
private Sprite2D _sprite2D;

public override void _Ready()
{
    base._Ready();

    _sprite2D = new Sprite2D(); // Create a new Sprite2D.
    AddChild(_sprite2D); // Add it as a child of this node.
}
```

To delete a node and free it from memory, you can call its `queue_free()` method. Doing so queues the node for deletion at the end of the current frame after it has finished processing. At that point, the engine removes the node from the scene and frees the object in memory.

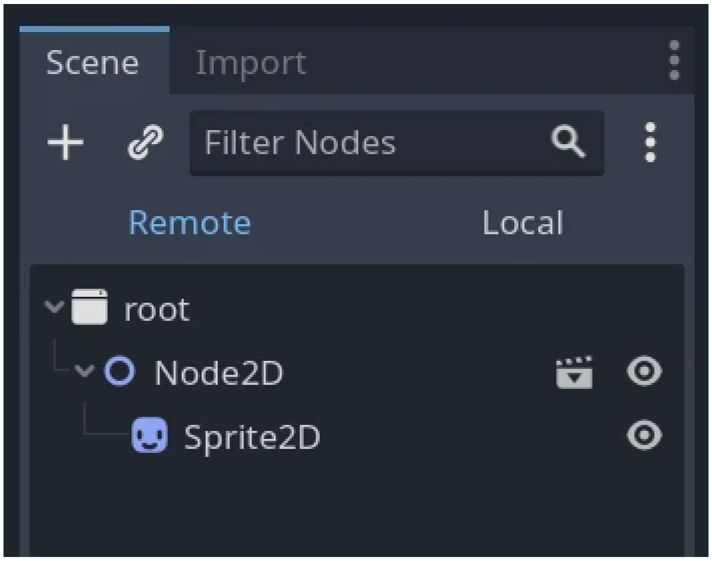
GDScript

```
sprite2d.queue_free()
```

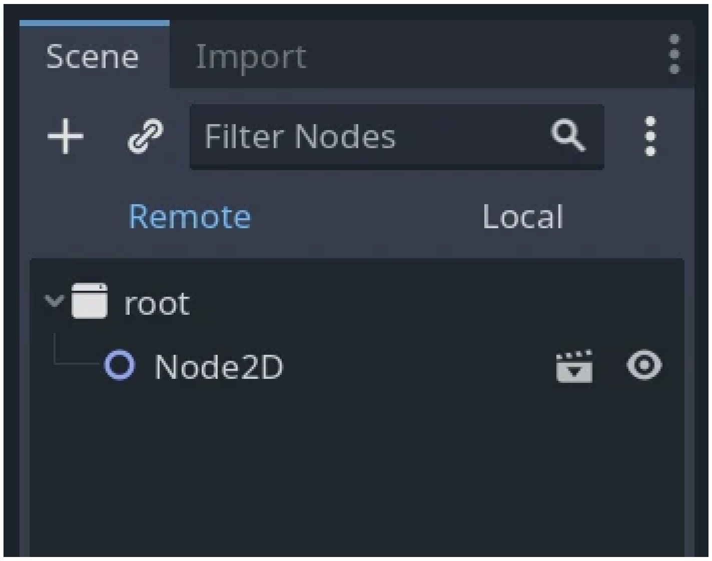
C#

```
_sprite2D.QueueFree();
```

Before calling `sprite2d.queue_free()`, the remote scene tree looks like this.


Scene
Import
Filter Nodes
Remote
Local
root
Node2D
Sprite2D

After the engine freed the node, the remote scene tree doesn't display the sprite anymore.


Scene
Import
Filter Nodes
Remote
Local
root
Node2D

You can alternatively call `free()` to immediately destroy the node. You should do this with care as any reference to it will instantly become `null`. We recommend using `queue_free()` unless you know what you're doing.

When you free a node, it also frees all its children. Thanks to this, to delete an entire branch of the scene tree, you only have to free the topmost parent node.

Instancing scenes

Scenes are templates from which you can create as many reproductions as you'd like. This operation is called instancing, and doing it from code happens in two steps:

1. Loading the scene from the local drive.
2. Creating an instance of the loaded PackedScene resource.

GDScript

```
var scene = load("res://my_scene.tscn")
```

C#

```
var scene = GD.Load<PackedScene>("res://MyScene.tscn");
```

Preloading the scene can improve the user's experience as the load operation happens when the compiler reads the script and not at runtime. This feature is only available with GDScript.

GDScript

```
var scene = preload("res://my_scene.tscn")
```

At that point, `scene` is a packed scene resource, not a node. To create the actual node, you need to call PackedScene.instantiate(). It returns a tree of nodes that you can use as a child of your current node.

GDScript

```
var instance = scene.instantiate()
add_child(instance)
```

C#

```
var instance = scene.Instantiate();
AddChild(instance);
```

The advantage of this two-step process is you can keep a packed scene loaded and create new instances on the fly. For example, to quickly instance several enemies or bullets.

Overridable functions

Godot's Node class provides virtual functions you can override to update nodes every frame or on specific events, like when they enter the scene tree.

This document presents the ones you'll use most often.

See also

Under the hood, these functions rely on Godot's low-level notifications system. To learn more about it, see Godot notifications.

Two functions allow you to initialize and get nodes besides the class's constructor: `_enter_tree()` and `_ready()`.

When the node enters the Scene Tree, it becomes active and the engine calls its `_enter_tree()` method. That node's children may not be part of the active scene yet. As you can remove and re-add nodes to the scene tree, this function may be called multiple times throughout a node's lifetime.

Most of the time, you'll use `_ready()` instead. This function is called only once in a node's lifetime, after `_enter_tree()`. `_ready()` ensures that all children have entered the scene tree first, so you can safely call `get_node()` on them.

See also

To learn more about getting node references, read Nodes and scene instances.

Another related callback is `_exit_tree()`, which the engine calls every time a node is about to exit the scene tree. This can be when you call [Node.remove_child()](#) or when you free a node.

GDScript

```
# Called every time the node enters the scene tree.
func _enter_tree():
    pass

# Called when both the node and its children have entered the
scene tree.
func _ready():
    pass

# Called when the node is about to leave the scene tree, after
all its
# children received the _exit_tree() callback.
func _exit_tree():
    pass
```

C#

```
// Called every time the node enters the scene tree.
public override void _EnterTree()
{
    base._EnterTree();
}

// Called when both the node and its children have entered the
scene tree.
public override void _Ready()
{
    base._Ready();
}

// Called when the node is about to leave the scene tree, after
all its
// children.
public override void _ExitTree()
{
    base._ExitTree();
}
```

The two virtual methods `_process()` and `_physics_process()` allow you to update the node, every frame and every physics frame respectively. For more information, read the dedicated documentation: Idle and Physics Processing.

GDScript

```
# Called every frame.
func _process(delta):
    pass

# Called every physics frame.
func _physics_process(delta):
    pass
```

C#

```
public override void _Process(double delta)
{
    // Called every frame.
    base._Process(delta);
}

public override void _PhysicsProcess(double delta)
{
    // Called every physics frame.
    base._PhysicsProcess(delta);
}
```

Two more essential built-in node callback functions are Node._unhandled_input() and Node._input(), which you use to both receive and process individual input events. The `_unhandled_input()` method receives every key press, mouse click, etc. that have not been handled already in an `_input()` callback or in a user interface component. You want to use it for gameplay input in general. The `_input()` callback allows you to intercept and process input events before `_unhandled_input()` gets them.

To learn more about inputs in Godot, see the Input section.

GDScript

```
# Called once for every event.
func _unhandled_input(event):
    pass

# Called once for every event before _unhandled_input(),
allowing you to
# consume some events.
func _input(event):
    pass
```

C#

```
// Called once for every event.
public override void _UnhandledInput(InputEvent @event)
{
    base._UnhandledInput(@event);
}

// Called once for every event before _UnhandledInput(),
allowing you to
// consume some events.
public override void _Input(InputEvent @event)
{
    base._Input(@event);
}
```

There are some more overridable functions like Node._get_configuration_warnings(). Specialized node types provide more callbacks like CanvasItem._draw() to draw programmatically or Control._gui_input() to handle clicks and input on UI elements.

Cross-language scripting

Godot allows you to mix and match scripting languages to suit your needs. This means a single project can define nodes in both C# and GDScript. This page will go through the possible interactions between two nodes written in different languages.

The following two scripts will be used as references throughout this page.

GDScript

```
extends Node

var my_field: String = "foo"

signal my_signal

func print_node_name(node: Node) -> void:
    print(node.get_name())

func print_array(arr: Array) -> void:
    for element in arr:
        print(element)

func print_n_times(msg: String, n: int) -> void:
    for i in range(n):
        print(msg)

func my_signal_handler():
    print("The signal handler was called!")
```

C#

```
using Godot;

public partial class MyCSharpNode : Node
{
    public string myField = "bar";

    [Signal] public delegate void MySignalEventHandler();

    public void PrintNodeName(Node node)
    {
        GD.Print(node.Name);
    }

    public void PrintArray(string[] arr)
    {
        foreach (string element in arr)
        {
            GD.Print(element);
        }
    }

    public void PrintNTimes(string msg, int n)
    {
        for (int i = 0; i < n; ++i)
        {
            GD.Print(msg);
        }
    }

    public void MySignalHandler()
    {
        GD.Print("The signal handler was called!");
    }
}
```

Instantiating nodes

If you're not using nodes from the scene tree, you'll probably want to instantiate nodes directly from the code.

Instantiating C# nodes from GDScript

Using C# from GDScript doesn't need much work. Once loaded (see Classes as resources), the script can be instantiated with new().

```
var my_csharp_script = load("res://Path/To/MyCSharpNode.cs")
var my_csharp_node = my_csharp_script.new()
```

Warning

When creating `.cs` scripts, you should always keep in mind that the class Godot will use is the one named like the `.cs` file itself. If that class does not exist in the file, you'll see the following error: `` Invalid call. Nonexistent function `new` in base. ``

For example, MyCoolNode.cs should contain a class named MyCoolNode.

The C# class needs to derive a Godot class, for example `GodotObject`. Otherwise, the same error will occur.

You also need to check your `.cs` file is referenced in the project's `.csproj` file. Otherwise, the same error will occur.

Instantiating GDScript nodes from C#

From the C# side, everything work the same way. Once loaded, the GDScript can be instantiated with GDScript.New().

```
GDScript MyGDScript = GD.Load<GDScript>
("res://path/to/my_gd_script.gd");
GodotObject myGDScriptNode = (GodotObject)MyGDScript.New(); //
This is a GodotObject.
```

Here we are using an Object, but you can use type conversion like explained in Type conversion and casting.

Accessing fields

Accessing C# fields from GDScript

Accessing C# fields from GDScript is straightforward, you shouldn't have anything to worry about.

```
print(my_csharp_node.myField) # bar
my_csharp_node.myField = "BAR"
print(my_csharp_node.myField) # BAR
```

Accessing GDScript fields from C#

As C# is statically typed, accessing GDScript from C# is a bit more convoluted, you will have to use GodotObject.Get() and GodotObject.Set(). The first argument is the name of the field you want to access.

```
GD.Print(myGDScriptNode.Get("my_field")); // foo
myGDScriptNode.Set("my_field", "FOO");
GD.Print(myGDScriptNode.Get("my_field")); // FOO
```

Keep in mind that when setting a field value you should only use types the GDScript side knows about. Essentially, you want to work with built-in types as described in GDScript reference or classes extending Object.

Calling methods

Calling C# methods from GDScript

Again, calling C# methods from GDScript should be straightforward. The marshalling process will do its best to cast the arguments to match function signatures. If that's impossible, you'll see the following error: `Invalid call. Nonexistent function `FunctionName`.`

```
my_csharp_node.PrintNodeName(self) # myGDScriptNode
# my_csharp_node.PrintNodeName() # This line will fail.

my_csharp_node.PrintNTimes("Hello there!", 2) # Hello there!
Hello there!

my_csharp_node.PrintArray(["a", "b", "c"]) # a, b, c
my_csharp_node.PrintArray([1, 2, 3]) # 1, 2, 3
```

Calling GDScript methods from C#

To call GDScript methods from C# you'll need to use GodotObject.Call(). The first argument is the name of the method you want to call. The following arguments will be passed to said method.

```
myGDScriptNode.Call("print_node_name", this); // my_csharp_node
// myGDScriptNode.Call("print_node_name"); // This line will
fail silently and won't error out.

myGDScriptNode.Call("print_n_times", "Hello there!", 2); //
Hello there! Hello there!

string[] arr = new string[] { "a", "b", "c" };
myGDScriptNode.Call("print_array", arr); // a, b, c
myGDScriptNode.Call("print_array", new int[] { 1, 2, 3 }); //
1, 2, 3
// Note how the type of each array entry does not matter as
long as it can be handled by the marshaller.
```

Warning

As you can see, if the first argument of the called method is an array, you'll need to cast it as `object`. Otherwise, each element of your array will be treated as a single argument and the function signature won't match.

Connecting to signals

Connecting to C# signals from GDScript

Connecting to a C# signal from GDScript is the same as connecting to a signal defined in GDScript:

```
my_csharp_node.MySignal.connect(my_signal_handler)
```

Connecting to GDScript signals from C#

Connecting to a GDScript signal from C# only works with the `Connect` method because no C# static types exist for signals defined by GDScript:

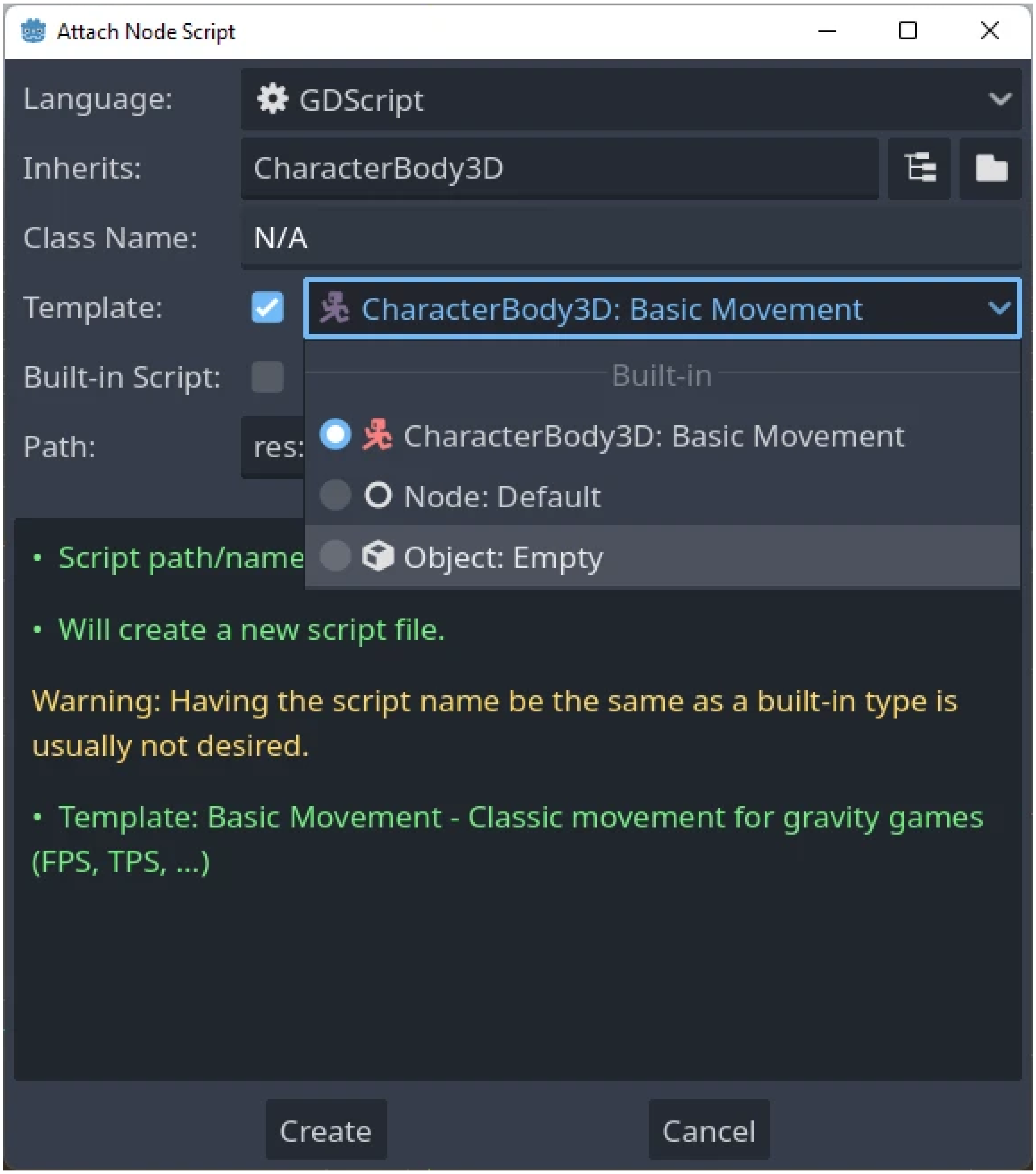
```
myGDScriptNode.Connect("my_signal",
Callable.From(MySignalHandler));
```

Inheritance

A GDScript file may not inherit from a C# script. Likewise, a C# script may not inherit from a GDScript file. Due to how complex this would be to implement, this limitation is unlikely to be lifted in the future. See this GitHub issue [https://github.com/godotengine/godot/issues/38352] for more information.

Creating script templates

Godot provides a way to use script templates as seen in the `Script Create Dialog` while creating a new script:



Attach Node Script

Language: GDScript

Inherits: CharacterBody3D

Class Name: N/A

Template: ☑ CharacterBody3D: Basic Movement

Built-in Script: ☐

Path: res:

Built-in

● CharacterBody3D: Basic Movement

○ Node: Default

○ Object: Empty

- Script path/name

- Will create a new script file.

Warning: Having the script name be the same as a built-in type is usually not desired.

- Template: Basic Movement - Classic movement for gravity games (FPS, TPS, ...)

A set of built-in script templates are provided with the editor, but it is also possible to create new ones and set them by default, both per project and at editor scope.

Templates are linked to a specific node type, so when you create a script you will only see the templates corresponding to that particular node, or one of its parent types. For example, if you are creating a script for a CharacterBody3D, you will only see templates defined for CharacterBody3Ds, Node3Ds or Nodes.

Locating the templates

There are two places where templates can be managed.

Editor-defined templates

These are available globally throughout any project. The location of these templates are determined per each OS:

- Windows: `%APPDATA%\Godot\script_templates\`
- Linux: `$HOME/.config/godot/script_templates/`
- macOS: `$HOME/Library/Application Support/Godot/script_templates/`

If you're getting Godot from somewhere other than the official website, such as Steam, the folder might be in a different location. You can find it using the Godot editor. Go to `Editor > Open Editor Data/Settings Folder` and it will open a folder in your file browser, inside that folder is the `script_templates` folder.

Project-defined templates

The default path to search for templates is the `res://script_templates/` directory. The path can be changed by configuring the `editor/script_templates_search_path` setting in the ProjectSettings, both via code and the editor.

If no `script_templates` directory is found within a project, it is simply ignored.

Template organization and naming

Both editor and project defined templates are organized in the following way:

```
template_path/node_type/file.extension
```

where:

- `template_path` is one of the 2 locations discussed in the previous two sections.
- `node_type` is the node it will apply to (for example, Node, or CharacterBody3D), This is **case-sensitive**. If a script isn't in the proper `node_type` folder, it won't be detected.
- `file` is the custom name you can chose for the template (for example, `platformer_movement` or `smooth_camera`).
- `extension` indicates which language the template will apply to (it should be `gd` for GDScript or `cs` for C#).

For example:

- `template_scripts/Node/smooth_camera.gd`
- `template_scripts/CharacterBody3D/platformer_movement.gd`

Default behaviour and overriding it

By default:

- the template's name is the same as the file name (minus the extension, prettyfied)
- the description is empty
- the space indent is set to 4
- the template will not be set as the default for the given node

It is possible to customize this behaviour by adding meta headers at the start of your file, like this:

GDScript

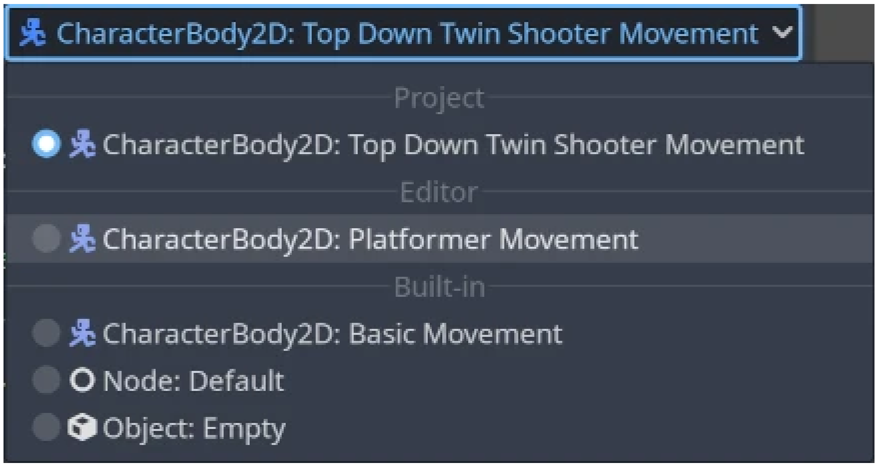
```
# meta-name: Platformer movement
# meta-description: Predefined movement for classical
platformers
# meta-default: true
# meta-space-indent: 4
```

C#

```
// meta-name: Platformer movement
// meta-description: Predefined movement for classical
platformers
// meta-default: true
// meta-space-indent: 4
```

In this case, the name will be set to "Platformer movement", with the given custom description, and it will be set as the default template for the node in which directory it has been saved.

This is an example of utilizing custom templates at editor and project level:



CharacterBody2D: Top Down Twin Shooter Movement
Project
CharacterBody2D: Top Down Twin Shooter Movement
Editor
CharacterBody2D: Platformer Movement
Built-in
CharacterBody2D: Basic Movement
Node: Default
Object: Empty

Note

The script templates have the same extension as the regular script files. This may lead to an issue of a script parser treating those templates as actual scripts within a project. To avoid this, make sure to ignore the directory containing them by creating an empty `.gdignore` file. The directory won't be visible throughout the project's filesystem anymore, yet the templates can be modified by an external text editor anytime.

Tip

By default, every C# file inside the project directory is included in the compilation. Script templates must be manually excluded from the C# project to avoid build errors. See Exclude files from the build [https://learn.microsoft.com/en-us/visualstudio/msbuild/how-to-exclude-files-from-the-build] in the Microsoft documentation.

It is possible to create editor-level templates that have the same level as a project-specific templates, and also that have the same name as a built-in one, all will be shown on the new script dialog.

Default template

To override the default template, create a custom template at editor or project level inside a `Node` directory (or a more specific type, if only a subtype wants to be overridden) and start the file with the `meta-default: true` header.

Only one template can be set as default at the same time for the same node type.

The `Default` templates for basic Nodes, for both GDScript and C#, are shown here so you can use these as the base for creating other templates:

GDScript

```
# meta-description: Base template for Node with default Godot
cycle methods

extends _BASE_


# Called when the node enters the scene tree for the first
time.
func _ready() -> void:
    pass # Replace with function body.


# Called every frame. 'delta' is the elapsed time since the
previous frame.
func _process(delta: float) -> void:
    pass
```

C#

```
// meta-description: Base template for Node with default Godot
cycle methods

using _BINDINGS_NAMESPACE_;
using System;

public partial class _CLASS_ : _BASE_
```

```
{
    // Called when the node enters the scene tree for the first time.
    public override void _Ready()
    {
    }

    // Called every frame. 'delta' is the elapsed time since the previous frame.
    public override void _Process(double delta)
    {
    }
}
```

The Godot editor provides a set of useful built-in node-specific templates, such as `basic_movement` for both CharacterBody2D and CharacterBody3D and `plugin` for EditorPlugin.

List of template placeholders

The following describes the complete list of built-in template placeholders which are currently implemented.

Base placeholders

Placeholder	**Description**
`_BINDINGS_NAMESPACE_`	The name of the Godot namespace (used in C# only).
`_CLASS_`	The name of the new class.

Placeholder	Description
`_BASE_`	The base type a new script inherits from.
`_TS_`	Indentation placeholder. The exact type and number of whitespace characters used for indentation is determined by the `text_editor/indent/type` and `text_editor/indent/size` settings in the EditorSettings respectively. Can be overridden by the `meta-space-indent` header on the template.

Type placeholders

There used to be, in Godot 3.x, placeholders for GDScript type hints that would get replaced whenever a template was used to create a new script, such as: `%INT_TYPE%`, `%STRING_TYPE%`, `%FLOAT_TYPE%` or `%VOID_RETURN%`.

The placeholders no longer work for Godot 4.x, but if the setting `text_editor/completion/add_type_hints` from EditorSettings is disabled, type hints for parameters and return types will be automatically removed for a few base types:

- `int`
- `String`
- `Array[String]`
- `float`
- `void`
- `:=` will be transformed into `=`

Evaluating expressions

Godot provides an Expression class you can use to evaluate expressions.

An expression can be:

- A mathematical expression such as `(2 + 4) * 16/4.0`.
- A built-in method call like `deg_to_rad(90)`.
- A method call on a user-provided script like `update_health()`, if `base_instance` is set to a value other than `null` when calling Expression.execute().

Note

The Expression class is independent from GDScript. It's available even if you compile Godot with the GDScript module disabled.

Basic usage

To evaluate a mathematical expression, use:

```
var expression = Expression.new()
expression.parse("20 + 10*2 - 5/2.0")
var result = expression.execute()
print(result)  # 37.5
```

The following operators are available:

Operator	**Notes**
Addition `+`	Can also be used to concatenate strings and arrays: - `"hello" + " world"` = `hello world` - `[1, 2] + [3, 4]` = `[1, 2, 3, 4]`
Subtraction (`-`)	
Multiplication (`*`)	
Division (`/`)	Performs and integer division if both operands are integers. If at least one of them is a floating-point number, returns a floating-point value.
Remainder (`%`)	Returns the remainder of an integer division (modulo). The result will always have the sign of the dividend.

Spaces around operators are optional. Also, keep in mind the usual order of operations [https://en.wikipedia.org/wiki/Order_of_operations] applies. Use parentheses to override the order of operations if needed.

All the Variant types supported in Godot can be used: integers, floating-point numbers, strings, arrays, dictionaries, colors, vectors, …

Arrays and dictionaries can be indexed like in GDScript:

```
# Returns 1.
[1, 2][0]

# Returns 3. Negative indices can be used to count from the end
of the array.
[1, 3][-1]

# Returns "green".
{"favorite_color": "green"}["favorite_color"]

# All 3 lines below return 7.0 (Vector3 is floating-point).
Vector3(5, 6, 7)[2]
Vector3(5, 6, 7)["z"]
Vector3(5, 6, 7).z
```

Passing variables to an expression

You can pass variables to an expression. These variables will then become available in the expression's "context" and will be substituted when used in the expression:

```
var expression = Expression.new()
# Define the variable names first in the second parameter of
`parse()`.
# In this example, we use `x` for the variable name.
expression.parse("20 + 2 * x", ["x"])
# Then define the variable values in the first parameter of
`execute()`.
# Here, `x` is assigned the integer value 5.
var result = expression.execute([5])
print(result)  # 30
```

Both the variable names and variable values **must** be specified as an array, even if you only define one variable. Also, variable names are **case-sensitive**.

Setting a base instance for the expression

By default, an expression has a base instance of `null`. This means the expression has no base instance associated to it.

When calling Expression.execute(), you can set the value of the `base_instance` parameter to a specific object instance such as `self`, another script instance or even a singleton:

```
func double(number):
    return number * 2


func _ready():
    var expression = Expression.new()
    expression.parse("double(10)")

    # This won't work since we're not passing the current
script as the base instance.
    var result = expression.execute([], null)
    print(result)  # null

    # This will work since we're passing the current script
(i.e. self)
    # as the base instance.
    result = expression.execute([], self)
    print(result)  # 20
```

Associating a base instance allows doing the following:

- Reference the instance's constants (`const`) in the expression.
- Reference the instance's member variables (`var`) in the expression.
- Call methods defined in the instance and use their return values in the expression.

Warning

Setting a base instance to a value other than `null` allows referencing constants, member variables, and calling all methods defined in the script attached to the instance. Allowing users to enter expressions may allow cheating in your game, or may even introduce security vulnerabilities if you allow arbitrary clients to run expressions on other players' devices.

Example script

The script below demonstrates what the Expression class is capable of:

```
const DAYS_IN_YEAR = 365
var script_member_variable = 1000


func _ready():
    # Constant mathexpression.
    evaluate("2 + 2")
    # Math expression with variables.
    evaluate("x + y", ["x", "y"], [60, 100])

    # Call built-in method (built-in math function call).
    evaluate("deg_to_rad(90)")

    # Call user method (defined in the script).
    # We can do this because the expression execution is bound
to `self`
    # in the `evaluate()` method.
    # Since this user method returns a value, we can use it in
math expressions.
    evaluate("call_me() + DAYS_IN_YEAR +
script_member_variable")
    evaluate("call_me(42)")
    evaluate("call_me('some string')")
```

```
func evaluate(command, variable_names = [], variable_values =
[]) -> void:
    var expression = Expression.new()
    var error = expression.parse(command, variable_names)
    if error != OK:
        push_error(expression.get_error_text())
        return

    var result = expression.execute(variable_values, self)

    if not expression.has_execute_failed():
        print(str(result))


func call_me(argument = null):
    print("\nYou called 'call_me()' in the expression text.")
    if argument:
        print("Argument passed: %s" % argument)

    # The method's return value is also the expression's return
value.
    return 0
```

The output from the script will be:

```
4
160
1.5707963267949

You called 'call_me()' in the expression text.
1365

You called 'call_me()' in the expression text.
Argument passed: 42
0

You called 'call_me()' in the expression text.
Argument passed: some string
0
```

Built-in functions

All methods in the Global Scope are available in the Expression class, even if no base instance is bound to the expression. The same parameters and return types are available.

However, unlike GDScript, parameters are **always required** even if they're specified as being optional in the class reference. In contrast, this restriction on arguments doesn't apply to user-made functions when you bind a base instance to the expression.

Change scenes manually

Sometimes it helps to have more control over how you swap scenes around. A Viewport's child nodes will render to the image it generates. This holds true even for nodes outside of the "current" scene. Autoloads fall into this category, and also scenes which you instantiate and add to the tree at runtime:

GDScript

```
var simultaneous_scene =
preload("res://levels/level2.tscn").instantiate()

func _add_a_scene_manually():
    # This is like autoloading the scene, only
    # it happens after already loading the main scene.
    get_tree().root.add_child(simultaneous_scene)
```

C#

```
public Node simultaneousScene;

public MyClass()
{
    simultaneousScene = ResourceLoader.Load<PackedScene>
("res://levels/level2.tscn").Instantiate();
}

public void _AddASceneManually()
{
    // This is like autoloading the scene, only
    // it happens after already loading the main scene.
    GetTree().Root.AddChild(simultaneousScene);
}
```

To complete the cycle and swap out the new scene with the old one, you have a choice to make. Many strategies exist for removing a scene from view of the Viewport. The tradeoffs involve balancing operation speed and memory consumption, as well as balancing data access and integrity.

1. **Delete the existing scene.** SceneTree.change_scene_to_file() and SceneTree.change_scene_to_packed() will delete the current scene immediately. You can also delete the main scene. Assuming the root node's name is "Main", you could do `get_node("/root/Main").free()` to delete the whole scene.
    - Unloads memory.
        - Pro: RAM is no longer dragging the dead weight.
        - Con: Returning to that scene is now more expensive since it must be loaded back into memory again (takes time AND memory). Not a problem if returning soon is unnecessary.
        - Con: No longer have access to that scene's data. Not a problem if using that data soon is unnecessary.
        - Note: It can be useful to preserve the data in a soon-to-be-deleted scene by re-attaching one or more of its nodes to a different scene, or even directly to the SceneTree.
    - Processing stops.
        - Pro: No nodes means no processing, physics processing, or input handling. The CPU is available to work on the new scene's contents.
        - Con: Those nodes' processing and input handling no longer operate. Not a problem if using the updated data is unnecessary.

2. **Hide the existing scene.** By changing the visibility or collision detection of the nodes, you can hide the entire node sub-tree from the player's perspective.
    - Memory still exists.
        - Pro: You can still access the data if needed.
        - Pro: There's no need to move any more nodes around to save data.
        - Con: More data is being kept in memory, which will be become a problem on memory-sensitive platforms like web or mobile.
    - Processing continues.
        - Pro: Data continues to receive processing updates, so the scene will keep any data within it that relies on delta time or frame data updated.
        - Pro: Nodes are still members of groups (since groups belong to the SceneTree).
        - Con: The CPU's attention is now divided between both scenes. Too much load could result in low frame rates. You should be sure to test performance as you go to ensure the target platform can support the load from this approach.
3. **Remove the existing scene from the tree.** Assign a variable to the existing scene's root node. Then use Node.remove_child(Node) to detach the entire scene from the tree.
    - Memory still exists (similar pros/cons as hiding it from view).
    - Processing stops (similar pros/cons as deleting it completely).

  - Pro: This variation of "hiding" it is much easier to show/hide. Rather than potentially keeping track of multiple changes to the scene, you only need to call the add/remove_child methods. This is similar to disabling game objects in other engines.
  - Con: Unlike with hiding it from view only, the data contained within the scene will become stale if it relies on delta time, input, groups, or other data that is derived from SceneTree access.

There are also cases where you may wish to have many scenes present at the same time, such as adding your own singleton at runtime, or preserving a scene's data between scene changes (adding the scene to the root node).

GDScript

```
get_tree().root.add_child(scene)
```

C#

```
GetTree().Root.AddChild(scene);
```

Another case may be displaying multiple scenes at the same time using SubViewportContainers. This is optimal for rendering different content in different parts of the screen (e.g. minimaps, split-screen multiplayer).

Each option will have cases where it is best appropriate, so you must examine the effects of each approach, and determine what path best fits your unique situation.

Instancing with signals

Signals provide a way to decouple game objects, allowing you to avoid forcing a fixed arrangement of nodes. One sign that a signal might be called for is when you find yourself using `get_parent()`. Referring directly to a node's parent means that you can't easily move that node to another location in the scene tree. This can be especially problematic when you are instancing objects at runtime and may want to place them in an arbitrary location in the running scene tree.

Below we'll consider an example of such a situation: firing bullets.

Shooting example

Consider a player character that can rotate and shoot towards the mouse. Every time the mouse button is clicked, we create an instance of the bullet at the player's location. See Creating instances for details.

We'll use an `Area2D` for the bullet, which moves in a straight line at a given velocity:

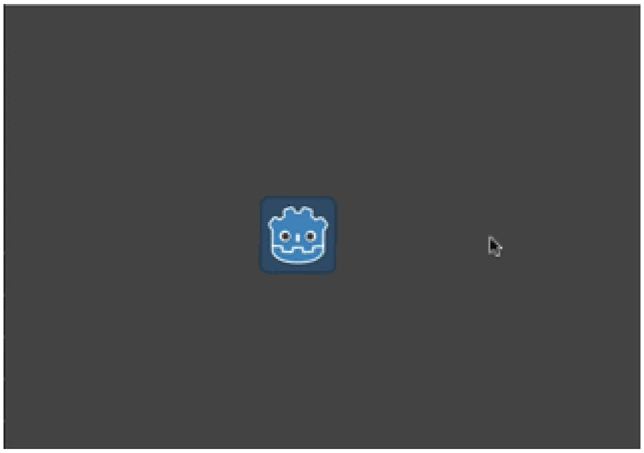
GDScript

```
extends Area2D

var velocity = Vector2.RIGHT

func _physics_process(delta):
    position += velocity * delta
```

C#

```
using Godot;

public partial class Bullet : Area2D
{
    public Vector2 Velocity { get; set; } = Vector2.Right;

    public override void _PhysicsProcess(double delta)
    {
        Position += Velocity * (float)delta;
    }
}
```

However, if the bullets are added as children of the player, then they will remain "attached" to the player as it rotates:



Instead, we need the bullets to be independent of the player's movement - once fired, they should continue traveling in a straight line and the player can no longer affect them. Instead of being added to the scene tree as a child of the player, it makes more sense to add the bullet as a child of the "main" game scene, which may be the player's parent or even further up the tree.

You could do this by adding the bullet to the main scene directly:

GDScript

```
var bullet_instance = Bullet.instantiate()
get_parent().add_child(bullet_instance)
```

C#

```
Node bulletInstance = Bullet.Instantiate();
GetParent().AddChild(bulletInstance);
```

However, this will lead to a different problem. Now if you try to test your "Player" scene independently, it will crash on shooting, because there is no parent node to access. This makes it a lot harder to test your player code independently and also means that if you decide to change your main scene's node structure, the player's parent may no longer be the appropriate node to receive the bullets.

The solution to this is to use a signal to "emit" the bullets from the player. The player then has no need to "know" what happens to the bullets after that - whatever node is connected to the signal can "receive" the bullets and take the appropriate action to spawn them.

Here is the code for the player using signals to emit the bullet:

GDScript

```
extends Sprite2D

signal shoot(bullet, direction, location)

var Bullet = preload("res://bullet.tscn")

func _input(event):
    if event is InputEventMouseButton:
        if event.button_index == MOUSE_BUTTON_LEFT and
event.pressed:
            shoot.emit(Bullet, rotation, position)

func _process(delta):
    look_at(get_global_mouse_position())
```

C#

```
using Godot;

```

```
public partial class Player : Sprite2D
{
    [Signal]
    public delegate void ShootEventHandler(PackedScene bullet,
float direction, Vector2 location);

    private PackedScene _bullet = GD.Load<PackedScene>
("res://Bullet.tscn");

    public override void _Input(InputEvent @event)
    {
        if (@event is InputEventMouseButton mouseButton)
        {
            if (mouseButton.ButtonIndex == MouseButton.Left &&
mouseButton.Pressed)
            {
                EmitSignal(SignalName.Shoot, _bullet, Rotation,
Position);
            }
        }
    }

    public override void _Process(double delta)
    {
        LookAt(GetGlobalMousePosition());
    }
}
```

In the main scene, we then connect the player's signal (it will appear in the "Node" tab of the Inspector)

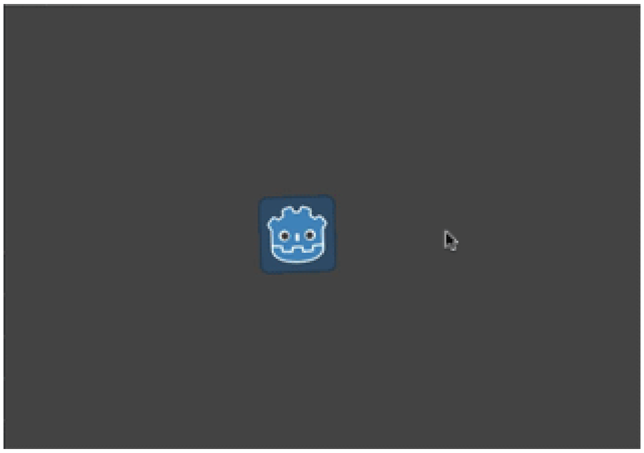
GDScript

```
func _on_player_shoot(Bullet, direction, location):
    var spawned_bullet = Bullet.instantiate()
    add_child(spawned_bullet)
    spawned_bullet.rotation = direction
    spawned_bullet.position = location
    spawned_bullet.velocity =
spawned_bullet.velocity.rotated(direction)
```

C#

```
private void OnPlayerShoot(PackedScene bullet, float direction,
Vector2 location)
{
    var spawnedBullet = bullet.Instantiate<Bullet>();
    AddChild(spawnedBullet);
    spawnedBullet.Rotation = direction;
    spawnedBullet.Position = location;
    spawnedBullet.Velocity =
spawnedBullet.Velocity.Rotated(direction);
}
```

Now the bullets will maintain their own movement independent of the player's rotation:



Pausing games and process mode

Introduction

In most games it is desirable to, at some point, interrupt the game to do something else, such as taking a break or changing options. Implementing a fine-grained control for what can be paused (and what cannot) is a lot of work, so a simple framework for pausing is provided in Godot.

How pausing works

To pause the game the pause state must be set. This is done by assigning `true` to the SceneTree.paused property:

GDScript

```
get_tree().paused = true
```

C#

```
GetTree().Paused = true;
```

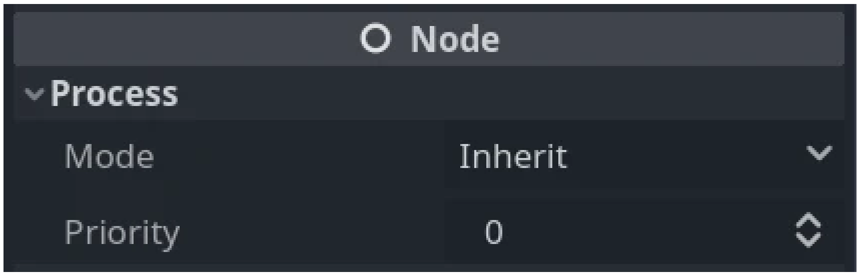
Doing this will cause two things. First, 2D and 3D physics will be stopped for all nodes. Second, the behavior of certain nodes will stop or start depending on their process mode.

Note

The physics servers can be made active while the game is paused by using their `set_active` methods.

Process Modes

Each node in Godot has a "Process Mode" that defines when it processes. It can be found and changed under a node's Node properties in the inspector.



Mode: Inherit

Priority: 0

You can also alter the property with code:

GDScript

```
func _ready():
    process_mode = Node.PROCESS_MODE_PAUSABLE
```

C#

```
public override void _Ready()
{
    ProcessMode = Node.ProcessModeEnum.Pausable;
}
```

This is what each mode tells a node to do:

- **Inherit**: Process depending on the state of the parent, grandparent, etc. The first parent that has a non-Inherit state.
- **Pausable**: Process the node (and its children in Inherit mode) only when the game is not paused.

- **WhenPaused**: Process the node (and its children in Inherit mode) *only* when the game is paused.
- **Always**: Process the node (and its children in Inherit mode) no matter what. Paused or not, this node will process.
- **Disabled**: The node (and its children in Inherit mode) will not process at all.

By default, all nodes have this property in the "Inherit" state. If the parent is set to "Inherit", then the grandparent will be checked and so on. If a state can't be found in any of the grandparents, the pause state in SceneTree is used. This means that, by default, when the game is paused every node will be paused. Several things happen when a node stops processing.

The `_process`, `_physics_process`, `_input`, and `_input_event` functions will not be called. However signals still work and cause their connected function to run, even if that function's script is attached to a node that is not currently being processed.

Animation nodes will pause their current animation, audio nodes will pause their current audio stream, and particles will pause. These resume automatically when the game is no longer paused.

It is important to note that even if a node is processing while the game is paused physics will **NOT** work for it by default. As stated earlier this is because the physics servers are turned off. The physics servers can be made active while the game is paused by using their `set_active` methods.

Pause menu example

Start by creating a button that will be used to pause the game.

Create a menu containing a close button, set the **Process Mode** of the menu's root node to **When Paused**, then hide the menu. Since the process mode is set to **When Paused** on the root node, all its children and grandchildren will inherit that process mode. This way, all the nodes in the menu will start processing when the game is paused.

Attach a script to the menu's root node, connect the pause button created earlier to a new method in the script, and inside that method pause the game and show the pause menu.

GDScript

```
func _on_pause_button_pressed():
    get_tree().paused = true
    show()
```

C#

```
private void OnPauseButtonPressed()
{
    GetTree().Paused = true;
    Show();
}
```

Finally, connect the menu's close button to a new method in the script. Inside that method, unpause the game and hide the pause menu.

GDScript

```
func _on_close_button_pressed():
    hide()
    get_tree().paused = false
```

C#

```
private void OnCloseButtonPressed()
{
    Hide();
    GetTree().Paused = false;
}
```

You should now have a working pause menu.

File system

Introduction

A file system manages how assets are stored and how they are accessed. A well-designed file system also allows multiple developers to edit the same source files and assets while collaborating. Godot stores all assets as files in its file system.

Implementation

The file system stores resources on disk. Anything, from a script, to a scene or a PNG image is a resource to the engine. If a resource contains properties that reference other resources on disk, the paths to those resources are also included. If a resource has sub-resources that are built-in, the resource is saved in a single file together with all the bundled sub-resources. For example, a font resource is often bundled together with the font textures.

The Godot file system avoids using metadata files. Existing asset managers and VCSs are better than anything we can implement, so Godot tries its best to play along with Subversion, Git, Mercurial, etc.

Example of file system contents:

```
/project.godot
/enemy/enemy.tscn
/enemy/enemy.gd
/enemy/enemysprite.png
/player/player.gd
```

project.godot

The `project.godot` file is the project description file, and it is always found at the root of the project. In fact, its location defines where the root is. This is the first file that Godot looks for when opening a project.

This file contains the project configuration in plain text, using the win.ini format. Even an empty `project.godot` can function as a basic definition of a blank project.

Path delimiter

Godot only supports `/` as a path delimiter. This is done for portability reasons. All operating systems support this, even Windows, so a path such as `C:\project\project.godot` needs to be typed as `C:/project/project.godot`.

Resource path

When accessing resources, using the host OS file system layout can be cumbersome and non-portable. To solve this problem, the special path `res://` was created.

The path `res://` will always point at the project root (where `project.godot` is located, so `res://project.godot` is always valid).

This file system is read-write only when running the project locally from the editor. When exported or when running on different devices (such as phones or consoles, or running from DVD), the file system will become read-only and writing will no longer be permitted.

User path

Writing to disk is still needed for tasks such as saving game state or downloading content packs. To this end, the engine ensures that there is a special path `user://` that is always writable. This path resolves differently depending on the OS the project is running on. Local path resolution is further explained in File paths in Godot projects.

Host file system

Alternatively host file system paths can also be used, but this is not recommended for a released product as these paths are not guaranteed to work on all platforms. However, using host file system paths can be useful when writing development tools in Godot.

Drawbacks

There are some drawbacks to this file system design. The first issue is that moving assets around (renaming them or moving them from one path to another inside the project) will break existing references to these assets. These references will have to be re-defined to point at the new asset location.

To avoid this, do all your move, delete and rename operations from within Godot, on the FileSystem dock. Never move assets from outside Godot, or dependencies will have to be fixed manually (Godot detects this and helps you fix them anyway, but why go the hard route?).

The second is that, under Windows and macOS, file and path names are case insensitive. If a developer working in a case insensitive host file system saves an asset as `myfile.PNG`, but then references it as `myfile.png`, it will work fine on their platform, but not on other platforms, such as Linux, Android, etc. This may also apply to exported binaries, which use a compressed package to store all files.

It is recommended that your team clearly define a naming convention for files when working with Godot. One fool-proof convention is to only allow lowercase file and path names.

Resources

Nodes and resources

Up to this tutorial, we focused on the Node class in Godot as that's the one you use to code behavior and most of the engine's features rely on it. There is another datatype that is just as important: Resource.

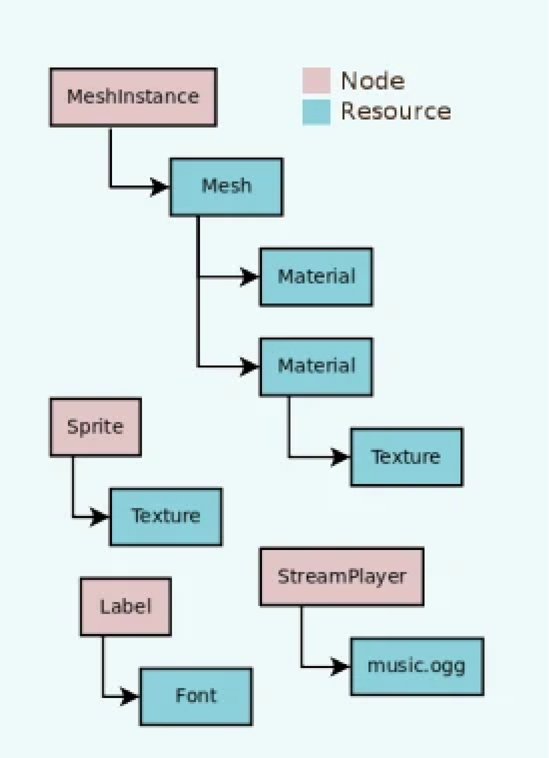
Nodes give you functionality: they draw sprites, 3D models, simulate physics, arrange user interfaces, etc. **Resources** are **data containers.** They don't do anything on their own: instead, nodes use the data contained in resources.

Anything Godot saves or loads from disk is a resource. Be it a scene (a `.tscn` or an `.scn` file), an image, a script... Here are some Resource examples:

- Texture
- Script
- Mesh
- Animation
- AudioStream
- Font
- Translation

When the engine loads a resource from disk, **it only loads it once**. If a copy of that resource is already in memory, trying to load the resource again will return the same copy every time. As resources only contain data, there is no need to duplicate them.

Every object, be it a Node or a Resource, can export properties. There are many types of Properties, like String, integer, Vector2, etc., and any of these types can become a resource. This means that both nodes and resources can contain resources as properties:

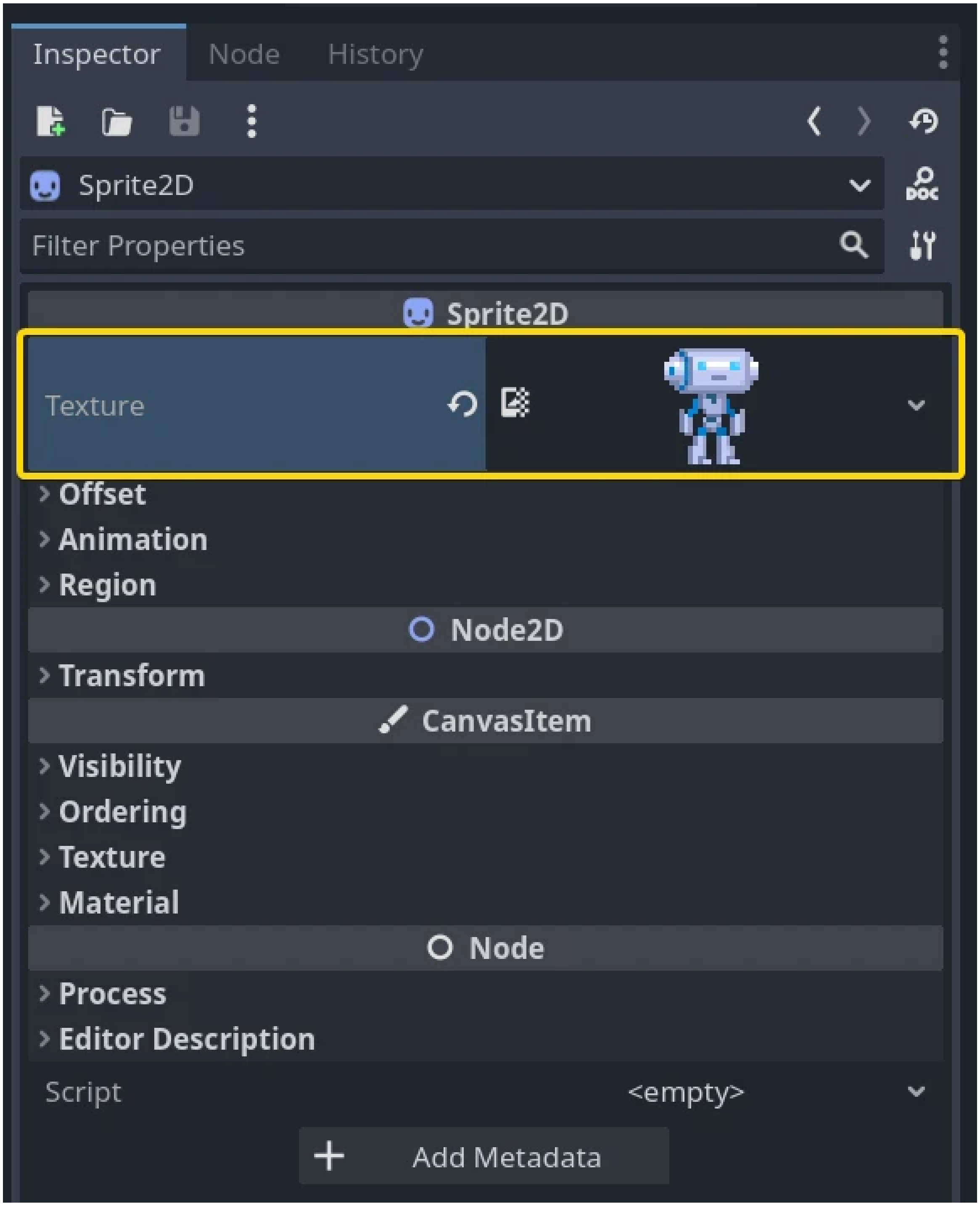

MeshInstance
Node
Resource
Mesh
Material
Material
Texture
Sprite
Texture
StreamPlayer
Label
music.ogg
Font

External vs built-in

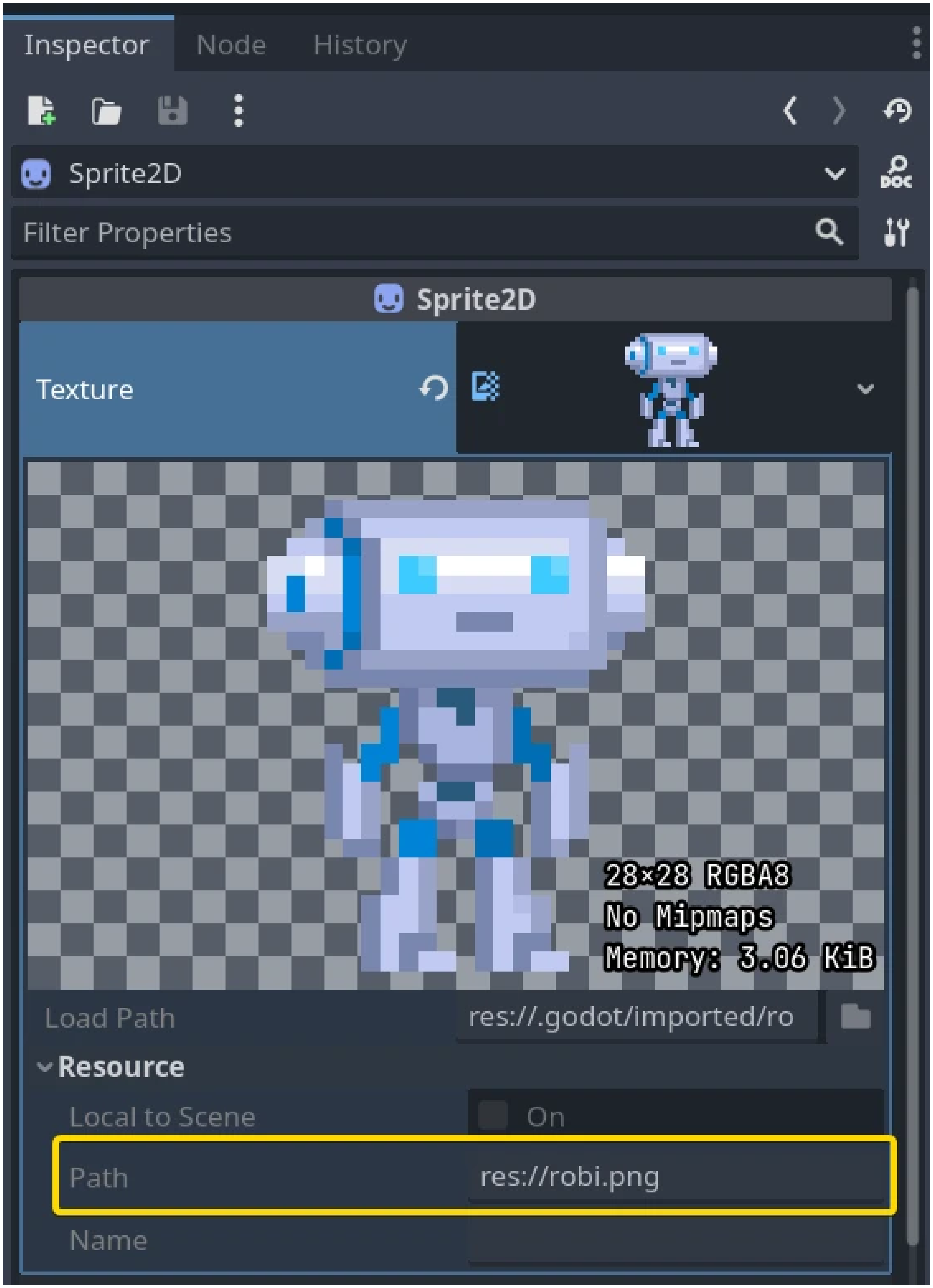
There are two ways to save resources. They can be:

1. **External** to a scene, saved on the disk as individual files.
2. **Built-in**, saved inside the `.tscn` or the `.scn` file they're attached to.

To be more specific, here's a Texture2D in a Sprite2D node:


Inspector
Node
History
Sprite2D
Filter Properties
Sprite2D
Texture
Offset
Animation
Region
Node2D
Transform
CanvasItem
Visibility
Ordering
Texture
Material
Node
Process
Editor Description
Script
<empty>
Add Metadata

Clicking the resource preview allows us to view the resource's properties.


Inspector
Node
History
Sprite2D
Filter Properties
Sprite2D
Texture
28×28 RGBA8
No Mipmaps
Memory: 3.06 KiB
Load Path
res://.godot/imported/ro
Resource
Local to Scene
On
Path
res://robi.png
Name

The path property tells us where the resource comes from. In this case, it comes from a PNG image called `robi.png`. When the resource comes from a file like this, it is an external resource. If you erase the path or this path is empty, it becomes a built-in resource.

The switch between built-in and external resources happens when you save the scene. In the example above, if you erase the path `"res://robi.png"` and save, Godot will save the image inside the `.tscn` scene file.

Note

Even if you save a built-in resource, when you instance a scene multiple times, the engine will only load one copy of it.

Loading resources from code

There are two ways to load resources from code. First, you can use the `load()` function anytime:

GDScript

```
func _ready():
    # Godot loads the Resource when it reads this very line.
    var imported_resource = load("res://robi.png")
    $sprite.texture = imported_resource
```

C#

```
public override void _Ready()
{
    // Godot loads the Resource when it executes this line.
    var texture = GD.Load<Texture>("res://Robi.png");
    var sprite = GetNode<Sprite2D>("sprite");
    sprite.Texture = texture;
}
```

You can also `preload` resources. Unlike `load`, this function will read the file from disk and load it at compile-time. As a result, you cannot call `preload` with a variable path: you need to use a constant string.

GDScript

```
func _ready():
    # Godot loads the resource at compile-time
    var imported_resource = preload("res://robi.png")
    get_node("sprite").texture = imported_resource
```

C#

```
// 'preload()' is unavailable in C Sharp.
```

Loading scenes

Scenes are also resources, but there is a catch. Scenes saved to disk are resources of type PackedScene. The scene is packed inside a Resource.

To get an instance of the scene, you have to use the PackedScene.instantiate() method.

GDScript

```
func _on_shoot():
        var bullet = preload("res://bullet.tscn").instantiate()
        add_child(bullet)
```

C#

```
private PackedScene _bulletScene = GD.Load<PackedScene>
("res://Bullet.tscn");

private void OnShoot()
{
    Node bullet = _bulletScene.Instantiate();
    AddChild(bullet);
}
```

This method creates the nodes in the scene's hierarchy, configures them, and returns the root node of the scene. You can then add it as a child of any other node.

The approach has several advantages. As the PackedScene.instantiate() function is fast, you can create new enemies, bullets, effects, etc. without having to load them again from disk each time. Remember that, as always, images, meshes, etc. are all shared between the scene instances.

Freeing resources

When a Resource is no longer in use, it will automatically free itself. Since, in most cases, Resources are contained in Nodes, when you free a node, the engine frees all the resources it owns as well if no other node uses them.

Creating your own resources

Like any Object in Godot, users can also script Resources. Resource scripts inherit the ability to freely translate between object properties and serialized text or binary data (*.tres, *.res). They also inherit the reference-counting memory management from the RefCounted type.

This comes with many distinct advantages over alternative data structures, such as JSON, CSV, or custom TXT files. Users can only import these assets as a Dictionary (JSON) or as a FileAccess to parse. What sets Resources apart is their inheritance of Object, RefCounted, and Resource features:

- They can define constants, so constants from other data fields or objects are not needed.
- They can define methods, including setter/getter methods for properties. This allows for abstraction and encapsulation of the underlying data. If the Resource script's structure needs to change, the game using the Resource need not also change.
- They can define signals, so Resources can trigger responses to changes in the data they manage.
- They have defined properties, so users know 100% that their data will exist.
- Resource auto-serialization and deserialization is a built-in Godot Engine feature. Users do not need to implement custom logic to import/export a resource file's data.
- Resources can even serialize sub-Resources recursively, meaning users can design even more sophisticated data structures.
- Users can save Resources as version-control-friendly text files (*.tres). Upon exporting a game, Godot serializes resource files as binary files (*.res) for increased speed and compression.
- Godot Engine's Inspector renders and edits Resource files out-of-the-box. As such, users often do not need to implement custom logic to visualize or edit their data. To do so, double-click the resource file in the FileSystem dock or click the folder icon in the Inspector and open the file in the dialog.
- They can extend **other** resource types besides just the base Resource.

Godot makes it easy to create custom Resources in the Inspector.

1. Create a plain Resource object in the Inspector. This can even be a type that derives Resource, so long as your script is extending that type.
2. Set the `script` property in the Inspector to be your script.

The Inspector will now display your Resource script's custom properties. If one edits those values and saves the resource, the

Inspector serializes the custom properties too! To save a resource from the Inspector, click the Inspector's tools menu (top right), and select "Save" or "Save As...".

If the script's language supports script classes, then it streamlines the process. Defining a name for your script alone will add it to the Inspector's creation dialog. This will auto-add your script to the Resource object you create.

Let's see some examples. Create a Resource and name it `bot_stats`. It should appear in your file tab with the full name `bot_stats.tres`. Without a script, it's useless, so let's add some data and logic! Attach a script to it named `bot_stats.gd` (or just create a new script, and then drag it to it).

Note

If you're using C#, you need to annotate your Resource class with the `[GlobalClass]` attribute for it to show up in the create resource GUI.

GDScript

```
extends Resource

@export var health: int
@export var sub_resource: Resource
@export var strings: PackedStringArray

# Make sure that every parameter has a default value.
# Otherwise, there will be problems with creating and editing
# your resource via the inspector.
func _init(p_health = 0, p_sub_resource = null, p_strings =
[]):
    health = p_health
    sub_resource = p_sub_resource
    strings = p_strings
```

C#

```
// BotStats.cs
using Godot;

namespace ExampleProject
{
    [GlobalClass]
    public partial class BotStats : Resource
    {
        [Export]
        public int Health { get; set; }

        [Export]
        public Resource SubResource { get; set; }

        [Export]
        public string[] Strings { get; set; }

        // Make sure you provide a parameterless constructor.
        // In C#, a parameterless constructor is different from a
        // constructor with all default values.
        // Without a parameterless constructor, Godot will have problems
        // creating and editing your resource via the inspector.
        public BotStats() : this(0, null, null) {}

        public BotStats(int health, Resource subResource, string[] strings)
        {
            Health = health;
            SubResource = subResource;
            Strings = strings ?? System.Array.Empty<string>();
        }
    }
}
```

Now, create a CharacterBody3D, name it `Bot`, and add the following script to it:

GDScript

```
extends CharacterBody3D

@export var stats: Resource

func _ready():
    # Uses an implicit, duck-typed interface for any 'health'-compatible resources.
    if stats:
        stats.health = 10
        print(stats.health)
        # Prints "10"
```

C#

```
// Bot.cs
using Godot;

namespace ExampleProject
{
    public partial class Bot : CharacterBody3D
    {
        [Export]
        public Resource Stats;

        public override void _Ready()
        {
            if (Stats is BotStats botStats)
            {
                GD.Print(botStats.Health); // Prints '10'.
            }
        }
    }
}
```

Now, select the CharacterBody3D node which we named `bot`, and drag&drop the `bot_stats.tres` resource onto the Inspector. It should print 10! Obviously, this setup can be used for more advanced features than this, but as long you really understand *how* it all worked, you should figure out everything else related to Resources.

Note

Resource scripts are similar to Unity's ScriptableObjects. The Inspector provides built-in support for custom resources. If desired though, users can even design their own Control-based tool scripts and combine them with an EditorPlugin to create custom visualizations and editors for their data.

Unreal Engine's DataTables and CurveTables are also easy to recreate with Resource scripts. DataTables are a String mapped to a custom struct, similar to a Dictionary mapping a String to a secondary custom Resource script.

GDScript

```gdscript
# bot_stats_table.gd
extends Resource

const BotStats = preload("bot_stats.gd")

var data = {
    "GodotBot": BotStats.new(10), # Creates instance with 10 health.
    "DifferentBot": BotStats.new(20) # A different one with 20 health.
}

func _init():
    print(data)
```

C#

```csharp
using Godot;

[GlobalClass]
public partial class BotStatsTable : Resource
{
    private Godot.Collections.Dictionary<string, BotStats> _stats = new Godot.Collections.Dictionary<string, BotStats>();

    public BotStatsTable()
    {
        _stats["GodotBot"] = new BotStats(10); // Creates
instance with 10 health.
        _stats["DifferentBot"] = new BotStats(20); // A
different one with 20 health.
        GD.Print(_stats);
    }
}
```

Instead of inlining the Dictionary values, one could also, alternatively:

1. Import a table of values from a spreadsheet and generate these key-value pairs.
2. Design a visualization within the editor and create a plugin that adds it to the Inspector when you open these types of Resources.

CurveTables are the same thing, except mapped to an Array of float values or a Curve/Curve2D resource object.

Warning

Beware that resource files (*.tres/*.res) will store the path of the script they use in the file. When loaded, they will fetch and load this script as an extension of their type. This means that trying to assign a subclass, i.e. an inner class of a script (such as using the `class` keyword in GDScript) won't work. Godot will not serialize the custom properties on the script subclass properly.

In the example below, Godot would load the `Node` script, see that it doesn't extend `Resource`, and then determine that the script failed to load for the Resource object since the types are incompatible.

GDScript

```
extends Node

class MyResource:
    extends Resource
    @export var value = 5

func _ready():
    var my_res = MyResource.new()

    # This will NOT serialize the 'value' property.
    ResourceSaver.save(my_res, "res://my_res.tres")
```

C#

```
using Godot;

public partial class MyNode : Node
{
    [GlobalClass]
    public partial class MyResource : Resource
    {
        [Export]
        public int Value { get; set; } = 5;
    }

    public override void _Ready()
    {
        var res = new MyResource();

        // This will NOT serialize the 'Value' property.
        ResourceSaver.Save(res, "res://MyRes.tres");
    }
}
```

Singletons (Autoload)

Introduction

Godot's scene system, while powerful and flexible, has a drawback: there is no method for storing information (e.g. a player's score or inventory) that is needed by more than one scene.

It's possible to address this with some workarounds, but they come with their own limitations:

- You can use a "master" scene that loads and unloads other scenes as its children. However, this means you can no longer run those scenes individually and expect them to work correctly.
- Information can be stored to disk in `user://` and then loaded by scenes that require it, but frequently saving and loading data is cumbersome and may be slow.

The Singleton pattern [https://en.wikipedia.org/wiki/Singleton_pattern] is a useful tool for solving the common use case where you need to store persistent information between scenes. In our case, it's possible to reuse the same scene or class for multiple singletons as long as they have different names.

Using this concept, you can create objects that:

- Are always loaded, no matter which scene is currently running.
- Can store global variables such as player information.
- Can handle switching scenes and between-scene transitions.
- *Act* like a singleton, since GDScript does not support global variables by design.

Autoloading nodes and scripts can give us these characteristics.

Note

Godot won't make an Autoload a "true" singleton as per the singleton design pattern. It may still be instanced more than once by the user if desired.

Tip

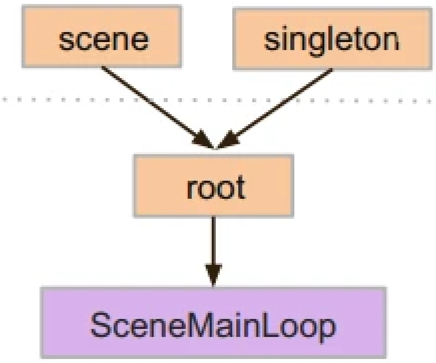
If you're creating an autoload as part of an editor plugin, consider registering it automatically in the Project Settings when the plugin is enabled.

Autoload

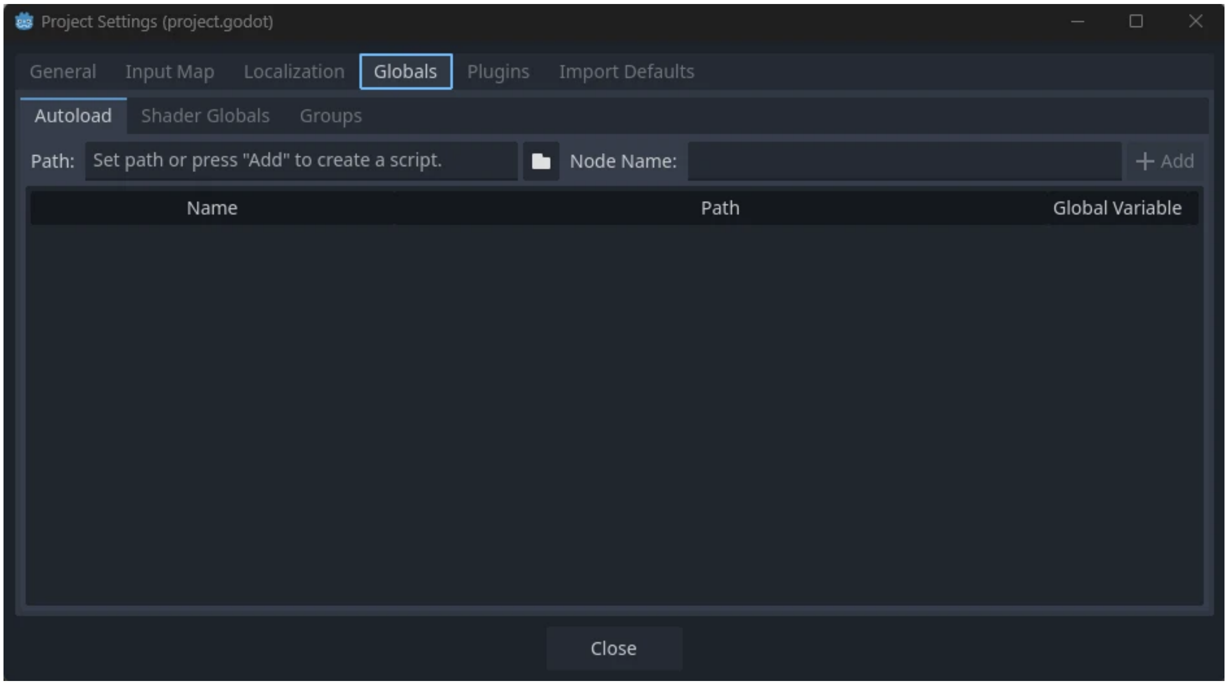
You can create an Autoload to load a scene or a script that inherits from Node.

Note

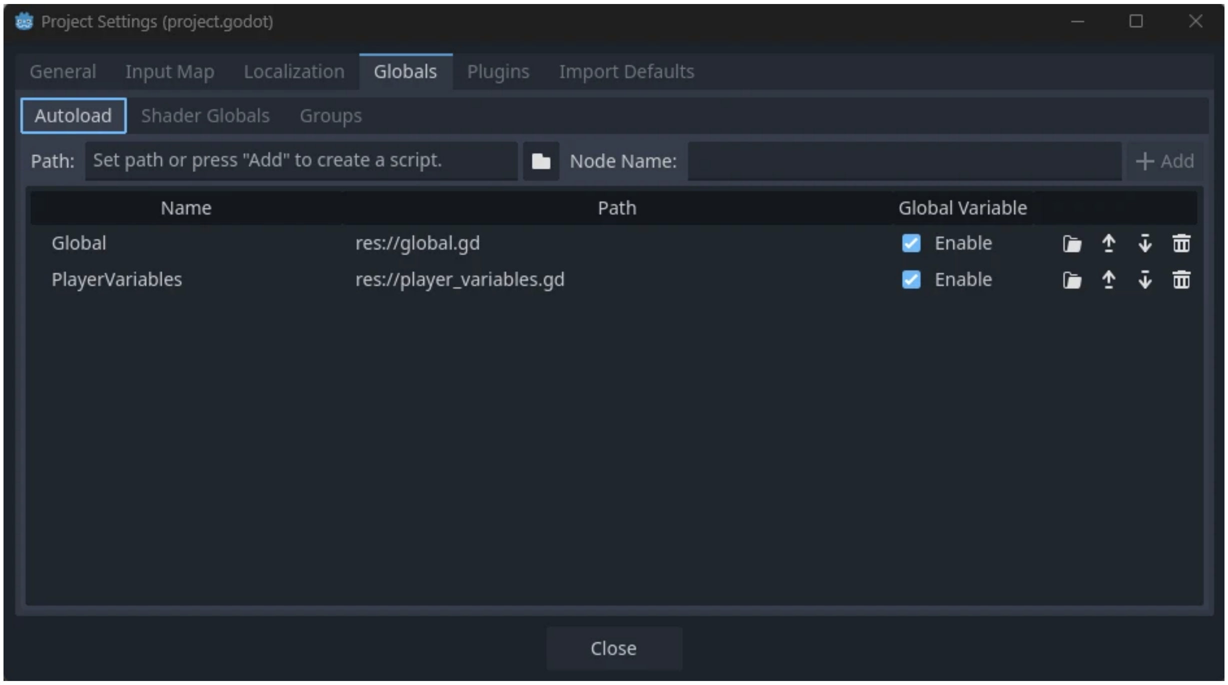
When autoloading a script, a Node will be created and the script will be attached to it. This node will be added to the root viewport before any other scenes are loaded.



To autoload a scene or script, start from the menu and navigate to **Project > Project Settings > Globals > Autoload**.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Autoload
Shader Globals
Groups
Path:
Set path or press "Add" to create a script.
Node Name:
Add
Name
Path
Global Variable
Close

Here you can add any number of scenes or scripts. Each entry in the list requires a name, which is assigned as the node's `name` property. The order of the entries as they are added to the global scene tree can be manipulated using the up/down arrow keys. Like regular scenes, the engine will read these nodes in top-to-bottom order.


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Autoload
Shader Globals
Groups
Path:
Set path or press "Add" to create a script.
Node Name:
Add
Close

Name	Path	Global Variable
Global	res://global.gd	☑ Enable
PlayerVariables	res://player_variables.gd	☑ Enable

If the **Enable** column is checked (which is the default), then the singleton can be accessed directly in GDScript:

GDScript

```
PlayerVariables.health -= 10
```

The **Enable** column has no effect in C# code. However, if the singleton is a C# script, a similar effect can be achieved by including a static property called `Instance` and assigning it in `_Ready()`:

C#

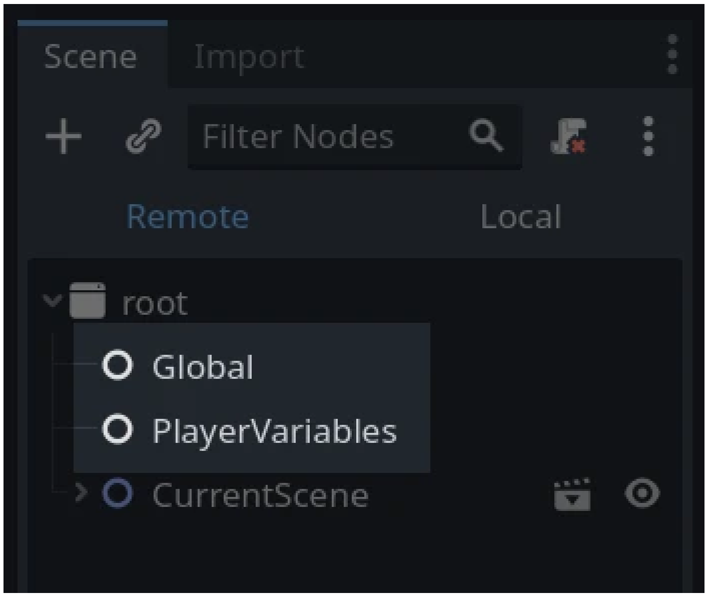
```
public partial class PlayerVariables : Node
{
    public static PlayerVariables Instance { get; private set;
}

    public int Health { get; set; }

    public override void _Ready()
    {
        Instance = this;
    }
}
```

This allows the singleton to be accessed from C# code without `GetNode()` and without a typecast:

C#

```
PlayerVariables.Instance.Health -= 10;
```

Note that autoload objects (scripts and/or scenes) are accessed just like any other node in the scene tree. In fact, if you look at the running scene tree, you'll see the autoloaded nodes appear:


Scene
Import
Filter Nodes
Remote
Local
root
Global
PlayerVariables
CurrentScene

Warning

Autoloads must **not** be removed using `free()` or `queue_free()` at runtime, or the engine will crash.

Custom scene switcher

This tutorial will demonstrate building a scene switcher using autoloads. For basic scene switching, you can use the SceneTree.change_scene_to_file() method (see Using SceneTree for details). However, if you need more complex behavior when changing scenes, this method provides more functionality.

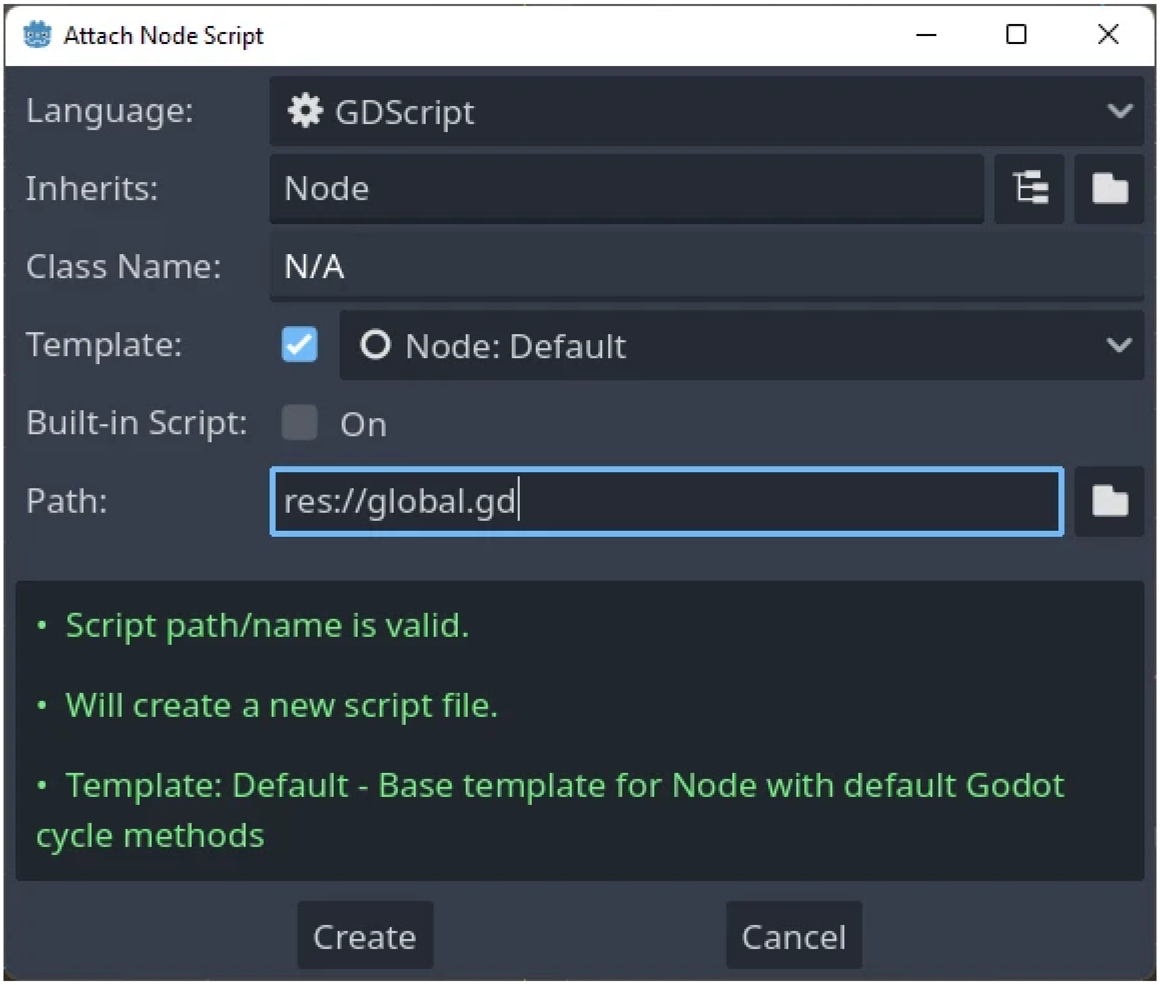
To begin, download the template from here: singleton_autoload_starter.zip [https://github.com/godotengine/godot-docs-project-starters/releases/download/latest-4.x/singleton_autoload_starter.zip] and open it in Godot.

A window notifying you that the project was last opened in an older Godot version may appear, that's not an issue. Click *Ok* to open the project.

The project contains two scenes: `scene_1.tscn` and `scene_2.tscn`. Each scene contains a label displaying the scene name and a button with its `pressed()` signal connected. When you run the project, it starts in `scene_1.tscn`. However, pressing the button does nothing.

Creating the script

Open the **Script** window and create a new script called `global.gd`. Make sure it inherits from `Node`:


Attach Node Script
Create
Cancel

Language: GDScript

Inherits: Node

Class Name: N/A

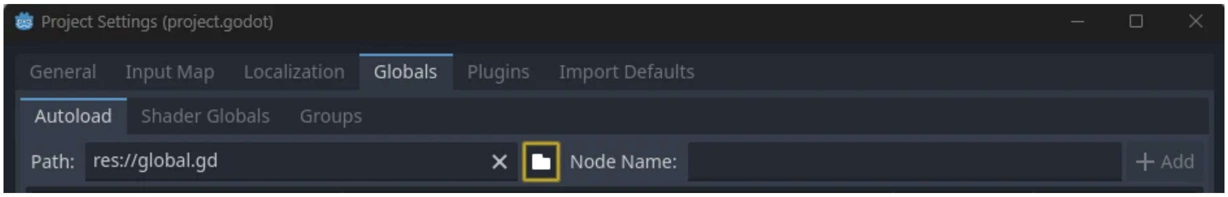
Template: ☑ Node: Default

Built-in Script: ☐ On

Path: res://global.gd

- Script path/name is valid.
- Will create a new script file.
- Template: Default - Base template for Node with default Godot cycle methods

The next step is to add this script to the autoLoad list. Starting from the menu, open **Project > Project Settings > Globals > Autoload** and select the script by clicking the browse button or typing its path: `res://global.gd`. Press **Add** to add it to the autoload list:


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Autoload
Shader Globals
Groups
+ Add

Path: res://global.gd

Node Name:

Now whenever we run any scene in the project, this script will always be loaded.

Returning to the script, it needs to fetch the current scene in the *_ready()* function. Both the current scene (the one with the button) and `global.gd` are children of root, but autoloaded nodes are always first. This means that the last child of root is always the loaded scene.

GDScript

```
extends Node

var current_scene = null

func _ready():
    var root = get_tree().root
    current_scene = root.get_child(root.get_child_count() - 1)
```

C#

```
using Godot;

public partial class Global : Node
{
    public Node CurrentScene { get; set; }

    public override void _Ready()
    {
        Viewport root = GetTree().Root;
        CurrentScene = root.GetChild(root.GetChildCount() - 1);
    }
}
```

Now we need a function for changing the scene. This function needs to free the current scene and replace it with the requested one.

GDScript

```
func goto_scene(path):
    # This function will usually be called from a signal
callback,
    # or some other function in the current scene.
    # Deleting the current scene at this point is
    # a bad idea, because it may still be executing code.
    # This will result in a crash or unexpected behavior.

    # The solution is to defer the load to a later time, when
    # we can be sure that no code from the current scene is
running:

    _deferred_goto_scene.call_deferred(path)


func _deferred_goto_scene(path):
    # It is now safe to remove the current scene.
    current_scene.free()

    # Load the new scene.
    var s = ResourceLoader.load(path)

    # Instance the new scene.
    current_scene = s.instantiate()

    # Add it to the active scene, as child of root.
    get_tree().root.add_child(current_scene)

    # Optionally, to make it compatible with the
SceneTree.change_scene_to_file() API.
    get_tree().current_scene = current_scene
```

C#

```
public void GotoScene(string path)
{
    // This function will usually be called from a signal
callback,
    // or some other function from the current scene.
    // Deleting the current scene at this point is
    // a bad idea, because it may still be executing code.
    // This will result in a crash or unexpected behavior.

    // The solution is to defer the load to a later time, when
    // we can be sure that no code from the current scene is
running:
    CallDeferred(MethodName.DeferredGotoScene, path);
}

public void DeferredGotoScene(string path)
{
    // It is now safe to remove the current scene.
    CurrentScene.Free();

    // Load a new scene.
    var nextScene = GD.Load<PackedScene>(path);

    // Instance the new scene.
    CurrentScene = nextScene.Instantiate();

    // Add it to the active scene, as child of root.
    GetTree().Root.AddChild(CurrentScene);

    // Optionally, to make it compatible with the
SceneTree.change_scene_to_file() API.
    GetTree().CurrentScene = CurrentScene;
}
```

Using Object.call_deferred(), the second function will only run once all code from the current scene has completed. Thus, the current scene will not be removed while it is still being used (i.e. its code is still running).

Finally, we need to fill the empty callback functions in the two scenes:

GDScript

```
# Add to 'scene_1.gd'.

func _on_button_pressed():
    Global.goto_scene("res://scene_2.tscn")
```

C#

```
// Add to 'Scene1.cs'.

private void OnButtonPressed()
{
    var global = GetNode<Global>("/root/Global");
    global.GotoScene("res://Scene2.tscn");
}
```

and

GDScript

```
# Add to 'scene_2.gd'.

func _on_button_pressed():
    Global.goto_scene("res://scene_1.tscn")
```

C#

```
// Add to 'Scene2.cs'.

private void OnButtonPressed()
{
    var global = GetNode<Global>("/root/Global");
    global.GotoScene("res://Scene1.tscn");
}
```

Run the project and test that you can switch between scenes by pressing the button.

Note

When scenes are small, the transition is instantaneous. However, if your scenes are more complex, they may take a noticeable amount of time to appear. To learn how to handle this, see the next tutorial: Background loading.

Alternatively, if the loading time is relatively short (less than 3 seconds or so), you can display a "loading plaque" by showing some kind of 2D element just before changing the scene. You can then hide it just after the scene is changed. This can be used to indicate to the player that a scene is being loaded.

Using SceneTree

Introduction

In previous tutorials, everything revolved around the concept of nodes. Scenes are collections of nodes. They become active once they enter the *scene tree*.

MainLoop

The way Godot works internally is as follows. There is the OS class, which is the only instance that runs at the beginning. Afterwards, all drivers, servers, scripting languages, scene system, etc are loaded.

When initialization is complete, OS needs to be supplied a MainLoop to run. Up to this point, all this is internals working (you can check main/main.cpp file in the source code if you are ever interested to see how this works internally).

The user program, or game, starts in the MainLoop. This class has a few methods, for initialization, idle (frame-synchronized callback), fixed (physics-synchronized callback), and input. Again, this is low level and when making games in Godot, writing your own MainLoop seldom makes sense.

SceneTree

One of the ways to explain how Godot works is that it's a high level game engine over a low level middleware.

The scene system is the game engine, while the OS and servers are the low level API.

The scene system provides its own main loop to OS, SceneTree. This is automatically instanced and set when running a scene, no need to do any extra work.

It's important to know that this class exists because it has a few important uses:

- It contains the root Viewport, to which a scene is added as a child when it's first opened to become part of the *Scene Tree* (more on that next).
- It contains information about the groups and has the means to call all nodes in a group or get a list of them.
- It contains some global state functionality, such as setting pause mode or quitting the process.

When a node is part of the Scene Tree, the SceneTree singleton can be obtained by calling Node.get_tree().

Root viewport

The root Viewport is always at the top of the scene. From a node, it can be obtained in two different ways:

GDScript

```
get_tree().root # Access via scene main loop.
get_node("/root") # Access via absolute path.
```

C#

```
GetTree().Root // Access via scene main loop.
GetNode("/root"); // Access via absolute path.
```

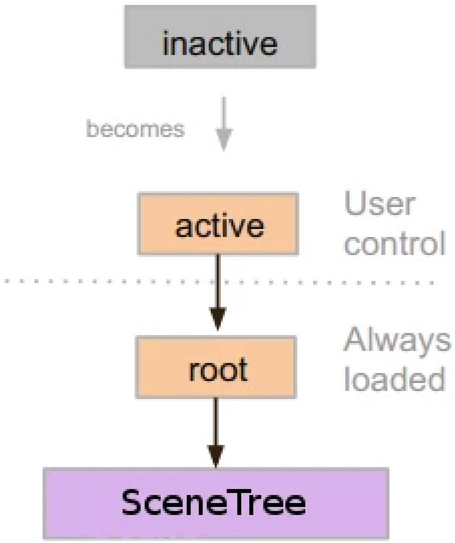
This node contains the main viewport. Anything that is a child of a Viewport is drawn inside of it by default, so it makes sense that the top of all nodes is always a node of this type otherwise nothing would be seen.

While other viewports can be created in the scene (for split-screen effects and such), this one is the only one that is never created by the user. It's created automatically inside SceneTree.

Scene tree

When a node is connected, directly or indirectly, to the root viewport, it becomes part of the *scene tree*.

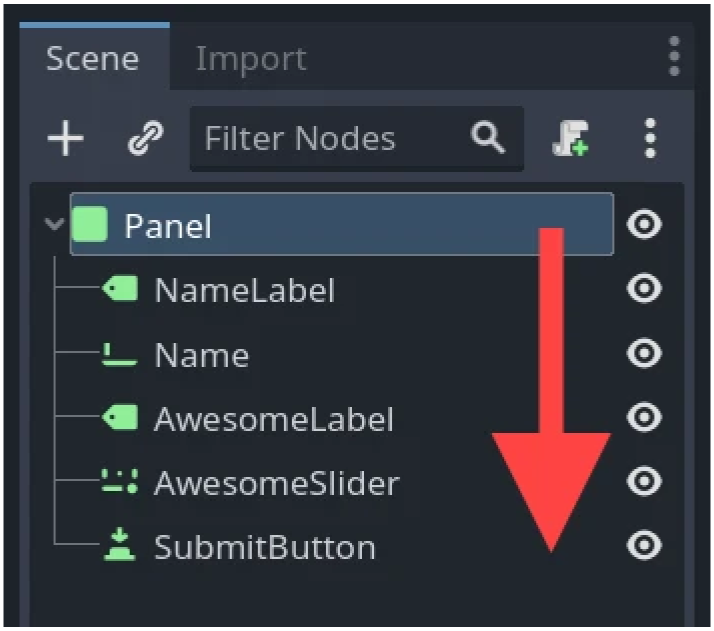
This means that as explained in previous tutorials, it will get the `_enter_tree()` and `_ready()` callbacks (as well as `_exit_tree()`).



When nodes enter the *Scene Tree*, they become active. They get access to everything they need to process, get input, display 2D and 3D visuals, receive and send notifications, play sounds, etc. When they are removed from the *scene tree*, they lose these abilities.

Tree order

Most node operations in Godot, such as drawing 2D, processing, or getting notifications are done in *tree order*, or top to bottom as seen in the editor (also known as pre-order traversal):



For example, the top node in a scene has its `_process()` function called first, then the node below it has its `_process()` function called, then the node below that and so on.

An important exception is the `_ready()` function: each parent node has its `_ready()` function called only after all its child nodes have their `_ready()` functions called, so that the parent knows its children are completely ready to be accessed. This is also known as post-order traversal. In the above image, `NameLabel` would be notified first (but only after its children, if it had any!), followed by `Name`, etc., and `Panel` would be notified last.

The order of operations can also be overridden using the `process_priority` node property. Nodes with a lower number are called first. For example, nodes with the priorities "0, 1, 2, 3" would be called in that order from left to right.

"Becoming active" by entering the *Scene Tree*

1. A scene is loaded from disk or created by scripting.
2. The root node of that scene (only one root, remember?) is added as either a child of the "root" Viewport (from SceneTree), or to any of its descendants.
3. Every node of the newly added scene will receive the "enter_tree" notification (`_enter_tree()` callback in GDScript) in top-to-bottom order (pre-order traversal).
4. Every node will receive the "ready" notification (`_ready()` callback in GDScript) for convenience, once all its children have received the "ready" notification (post-order traversal).
5. When a scene (or part of it) is removed, they receive the "exit scene" notification (`_exit_tree()` callback in GDScript) in bottom-to-top order (the exact reverse of top-to-bottom order).

Changing current scene

After a scene is loaded, you may want to change this scene for another one. One way to do this is to use the SceneTree.change_scene_to_file() function:

GDScript

```
func _my_level_was_completed():
    get_tree().change_scene_to_file("res://levels/level2.tscn")
```

C#

```
public void _MyLevelWasCompleted()
{
    GetTree().ChangeSceneToFile("res://levels/level2.tscn");
}
```

Rather than using file paths, one can also use ready-made PackedScene resources using the equivalent function SceneTree.change_scene_to_packed(PackedScene scene):

GDScript

```
var next_scene = preload("res://levels/level2.tscn")

func _my_level_was_completed():
    get_tree().change_scene_to_packed(next_scene)
```

C#

```
public void _MyLevelWasCompleted()
{
    var nextScene =
(PackedScene)ResourceLoader.Load("res://levels/level2.tscn");
    GetTree().ChangeSceneToPacked(nextScene);
}
```

These are quick and useful ways to switch scenes but have the drawback that the game will stall until the new scene is loaded and running. At some point in the development of your game, it may be preferable to create proper loading screens with progress bar, animated indicators or threaded (background) loading. This must be done manually using Singletons (Autoload) and Background loading.

Scene Unique Nodes

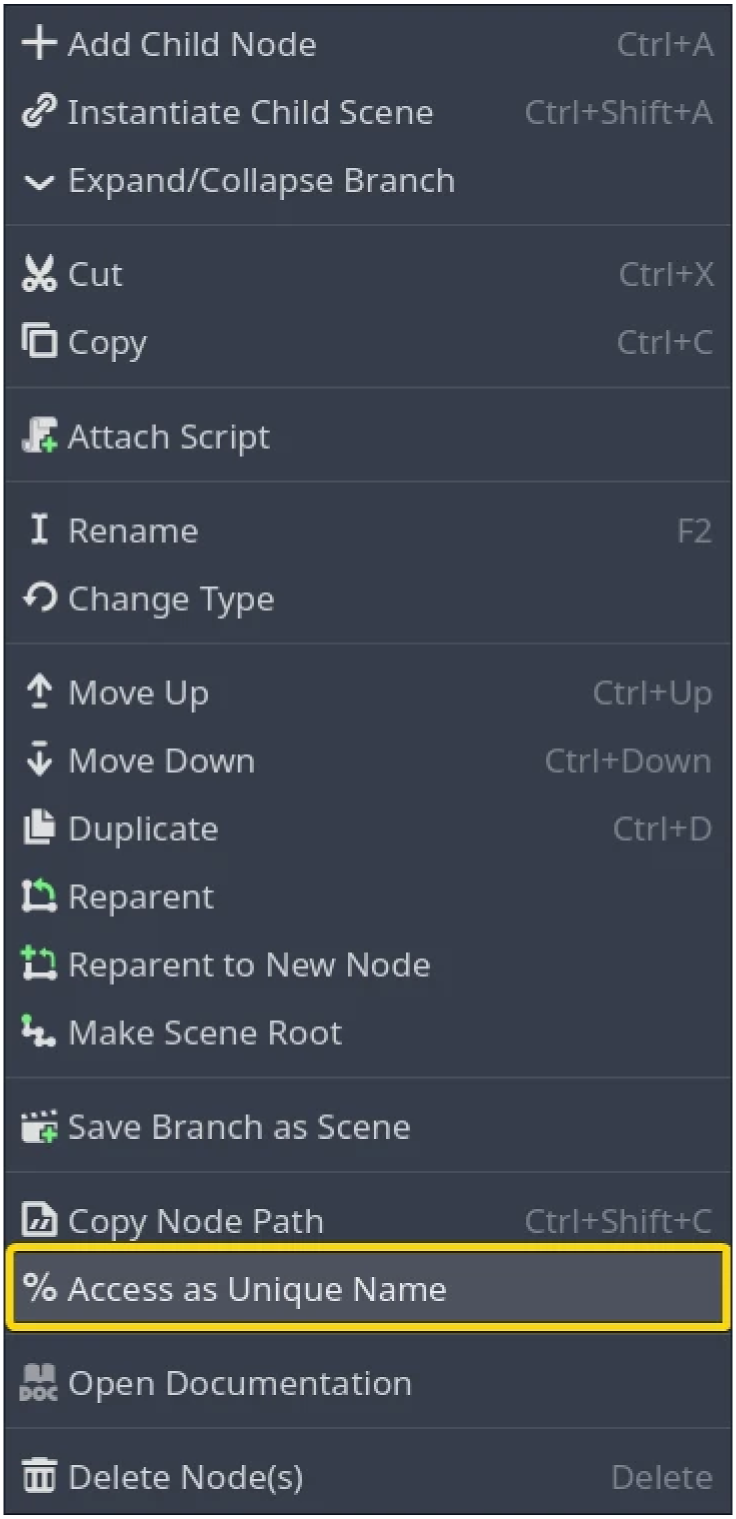
Introduction

Using `get_node()` to reference nodes from a script can sometimes be fragile. If you move a button in a UI scene from one panel to another, the button's node path changes, and if a script uses `get_node()` with a hard-coded node path, the script will not be able to find the button anymore.

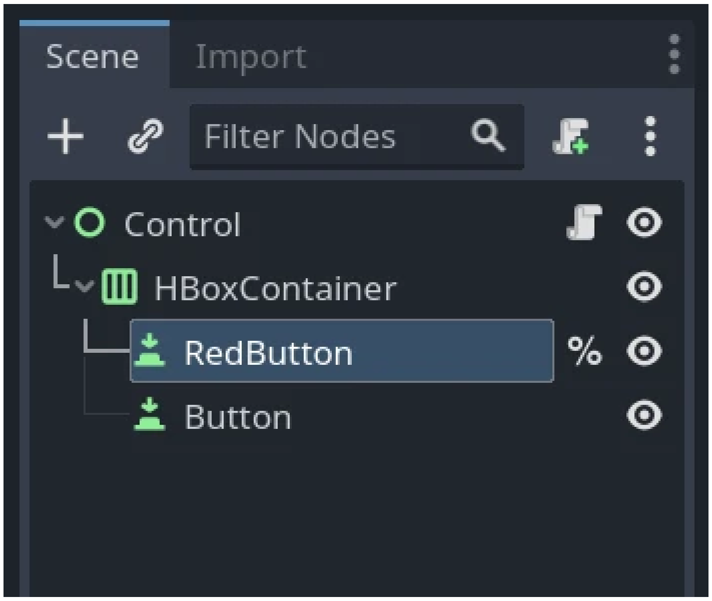
In situations like this, the node can be turned into a scene unique node to avoid having to update the script every time the node's path is changed.

Creation and usage

In the Scene tree dock, right-click on a node and select **Access as Unique Name** in the context menu.


Add Child Node Ctrl+A
Instantiate Child Scene Ctrl+Shift+A
Expand/Collapse Branch
Cut Ctrl+X
Copy Ctrl+C
Attach Script
Rename F2
Change Type
Move Up Ctrl+Up
Move Down Ctrl+Down
Duplicate Ctrl+D
Reparent
Reparent to New Node
Make Scene Root
Save Branch as Scene
Copy Node Path Ctrl+Shift+C
Access as Unique Name
Open Documentation
Delete Node(s) Delete

After selecting the option, the node will now have a percent symbol (**%**) next to its name in the scene tree:


Scene
Import
Filter Nodes
Control
HBoxContainer
RedButton
%
Button

You can now use the node in your script. For example, you can reference it with a `get_node()` method call by typing the % symbol, followed by the node's name:

GDScript

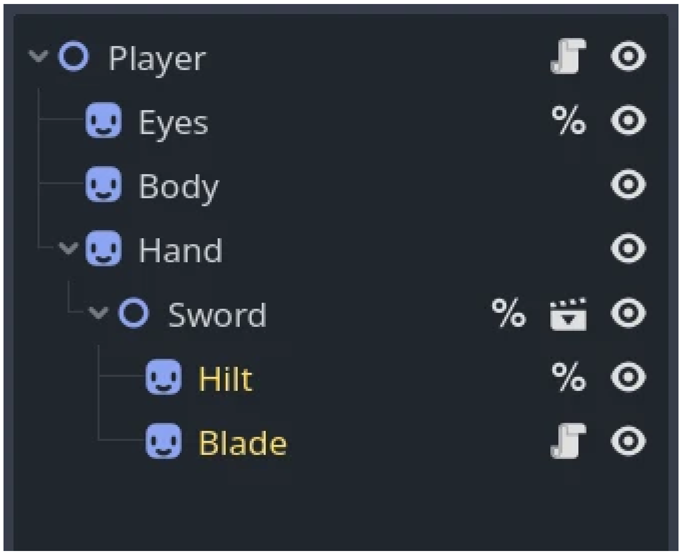
```
get_node("%RedButton").text = "Hello"
%RedButton.text = "Hello" # Shorter syntax
```

C#

```
GetNode<Button>("%RedButton").Text = "Hello";
```

Same-scene limitation

A scene unique node can only be retrieved by a node inside the same scene. To demonstrate this limitation, consider this example **Player** scene that instances a **Sword** scene:



Here are the results of `get_node()` calls inside the **Player** script:

- `get_node("%Eyes")` returns the **Eyes** node.
- `get_node("%Hilt")` returns `null`.

These are the results of `get_node()` calls inside the **Sword** script:

- `get_node("%Eyes")` returns `null`.
- `get_node("%Hilt")` returns the **Hilt** node.

If a script has access to a node in another scene, it can call `get_node()` on that node to get scene unique nodes from that node's scene. This also works in a node path, which avoids multiple `get_node()` calls. Here are two ways to get the **Hilt** node from the **Player** script using scene unique nodes:

- `get_node("Hand/Sword").get_node("%Hilt")` returns the **Hilt** node.
- `get_node("Hand/Sword/%Hilt")` also returns the **Hilt** node.

Scene unique names don't only work at the end of a node path. They can be used in the middle to navigate from one node to another. For example, the **Sword** node is marked as a scene unique node in the **Player** scene, so this is possible:

- `get_node("%Sword/%Hilt")` returns the **Hilt** node.

Alternatives

Scene unique nodes are a useful tool to navigate a scene. However, there are some situations where other techniques may be better.

A Group allows locating a node (or a group of many nodes) from any other node, no matter what scene the two nodes are located in.

A Singleton (Autoload) is an always loaded node that can be accessed directly by any node regardless of the scene. These are useful when some data or functionality is shared globally.

Node.find_child() finds a node by name without knowing its full path. This seems similar to a scene unique node, but this method is able to find nodes in nested scenes, and doesn't require marking the node in the scene editor in any way. However, this method is slow. Scene unique nodes are cached by Godot and are fast to retrieve, but each time the method is called, `find_child()` needs to check every descendant (every child, grandchild, and so on).

Shaders

- Introduction to shaders
- Shading reference
- Your first shader
- Shader materials
- Using VisualShaders
- Using compute shaders
- Screen-reading shaders
- Converting GLSL to Godot shaders
- Shaders style guide
- Using a SubViewport as a texture
- Custom post-processing
- Advanced post-processing
- Making trees

Introduction to shaders

This page explains what shaders are and will give you an overview of how they work in Godot. For a detailed reference of the engine's shading language, see Shading language.

Shaders are a special kind of program that runs on Graphics Processing Units (GPUs). They were initially used to shade 3D scenes but can nowadays do much more. You can use them to control how the engine draws geometry and pixels on the screen, allowing you to achieve all sorts of effects.

Modern rendering engines like Godot draw everything with shaders: graphics cards can run thousands of instructions in parallel, leading to incredible rendering speed.

Because of their parallel nature, though, shaders don't process information the way a typical program does. Shader code runs on each vertex or pixel in isolation. You cannot store data between frames either. As a result, when working with shaders, you need to code and think differently from other programming languages.

Suppose you want to update all the pixels in a texture to a given color. In GDScript, your code would use `for` loops:

```
for x in range(width):
  for y in range(height):
    set_color(x, y, some_color)
```

Your code is already part of a loop in a shader, so the corresponding code would look like this.

```
void fragment() {
  COLOR = some_color;
}
```

Note

The graphics card calls the `fragment()` function once or more for each pixel it has to draw. More on that below.

Shaders in Godot

Godot provides a shading language based on the popular OpenGL Shading Language (GLSL) but simplified. The engine handles some of the lower-level initialization work for you, making it easier to write complex shaders.

In Godot, shaders are made up of main functions called "processor functions". Processor functions are the entry point for your shader into the program. There are seven different processor functions.

1. The `vertex()` function runs over all the vertices in the mesh and sets their positions and some other per-vertex variables. Used in canvas_item shaders and spatial shaders.
2. The `fragment()` function runs for every pixel covered by the mesh. It uses values output by the `vertex()` function, interpolated between the vertices. Used in canvas_item shaders and spatial shaders.
3. The `light()` function runs for every pixel and for every light. It takes variables from the `fragment()` function and from its previous runs. Used in canvas_item shaders and spatial shaders.
4. The `start()` function runs for every particle in a particle system once when the particle is first spawned. Used in particles shaders.
5. The `process()` function runs for every particle in a particle system for each frame. Used in particles shaders.
6. The `sky()` function runs for every pixel in the radiance cubemap when the radiance cubemap needs to be updated, and for every

pixel on the current screen. Used in sky shaders.
7. The `fog()` function runs for every froxel in the volumetric fog froxel buffer that intersects with the FogVolume. Used by fog shaders.

Warning

The `light()` function won't run if the `vertex_lighting` render mode is enabled, or if **Rendering > Quality > Shading > Force Vertex Shading** is enabled in the Project Settings. It's enabled by default on mobile platforms.

Note

Godot also exposes an API for users to write totally custom GLSL shaders. For more information see Using compute shaders.

Shader types

Instead of supplying a general-purpose configuration for all uses (2D, 3D, particles, sky, fog), you must specify the type of shader you're writing. Different types support different render modes, built-in variables, and processing functions.

In Godot, all shaders need to specify their type in the first line, like so:

```
shader_type spatial;
```

Here are the available types:

- spatial for 3D rendering.
- canvas_item for 2D rendering.
- particles for particle systems.

- sky to render Skies.
- fog to render FogVolumes

Render modes

Shaders have optional render modes you can specify on the second line, after the shader type, like so:

```
shader_type spatial;
render_mode unshaded, cull_disabled;
```

Render modes alter the way Godot applies the shader. For example, the `unshaded` mode makes the engine skip the built-in light processor function.

Each shader type has different render modes. See the reference for each shader type for a complete list of render modes.

Vertex processor

The `vertex()` processing function is called once for every vertex in `spatial` and `canvas_item` shaders.

Each vertex in your world's geometry has properties like a position and color. The function modifies those values and passes them to the fragment function. You can also use it to send extra data to the fragment function using varyings.

By default, Godot transforms your vertex information for you, which is necessary to project geometry onto the screen. You can use render modes to transform the data yourself; see the Spatial shader doc for an example.

Fragment processor

The `fragment()` processing function is used to set up the Godot material parameters per pixel. This code runs on every visible pixel the object or primitive draws. It is only available in `spatial`, `canvas_item`, and `sky` shaders.

The standard use of the fragment function is to set up material properties used to calculate lighting. For example, you would set values for `ROUGHNESS`, `RIM`, or `TRANSMISSION`, which would tell the light function how the lights respond to that fragment. This makes it possible to control a complex shading pipeline without the user having to write much code. If you don't need this built-in functionality, you can ignore it and write your own light processing function, and Godot will optimize it away. For example, if you do not write a value to `RIM`, Godot will not calculate rim lighting. During compilation, Godot checks to see if `RIM` is used; if not, it cuts all the corresponding code out. Therefore, you will not waste calculations on the effects that you do not use.

Light processor

The `light()` processor runs per pixel too, and it runs once for every light that affects the object. It does not run if no lights affect the object. It exists as a function called inside the `fragment()` processor and typically operates on the material properties setup inside the `fragment()` function.

The `light()` processor works differently in 2D than it does in 3D; for a description of how it works in each, see their documentation, CanvasItem shaders and Spatial shaders, respectively.

Shading reference

- Shading language
- Shader preprocessor
- Spatial shaders
- CanvasItem shaders
- Particle shaders
- Sky shaders
- Fog shaders

Shading language

Introduction

Godot uses a shading language similar to GLSL ES 3.0. Most datatypes and functions are supported, and the few remaining ones will likely be added over time.

If you are already familiar with GLSL, the Godot Shader Migration Guide is a resource that will help you transition from regular GLSL to Godot's shading language.

Data types

Most GLSL ES 3.0 datatypes are supported:

Type	Description
void	Void datatype, useful only for functions that return nothing.
bool	Boolean datatype, can only contain `true` or `false`.
bvec2	Two-component vector of booleans.
bvec3	Three-component vector of booleans.

Type	Description
bvec4	Four-component vector of booleans.
int	32 bit signed scalar integer.
ivec2	Two-component vector of signed integers.
ivec3	Three-component vector of signed integers.
ivec4	Four-component vector of signed integers.
uint	Unsigned scalar integer; can't contain negative numbers.
uvec2	Two-component vector of unsigned integers.
uvec3	Three-component vector of unsigned integers.
uvec4	Four-component vector of unsigned integers.
float	32 bit floating-point scalar.
vec2	Two-component vector of floating-point values.

Type	**Description**
vec3	Three-component vector of floating-point values.
vec4	Four-component vector of floating-point values.
mat2	2x2 matrix, in column major order.
mat3	3x3 matrix, in column major order.
mat4	4x4 matrix, in column major order.
sampler2D	Sampler type for binding 2D textures, which are read as float.
isampler2D	Sampler type for binding 2D textures, which are read as signed integer.
usampler2D	Sampler type for binding 2D textures, which are read as unsigned integer.
sampler2DArray	Sampler type for binding 2D texture arrays, which are read as float.

Type	Description
isampler2DArray	Sampler type for binding 2D texture arrays, which are read as signed integer.
usampler2DArray	Sampler type for binding 2D texture arrays, which are read as unsigned integer.
sampler3D	Sampler type for binding 3D textures, which are read as float.
isampler3D	Sampler type for binding 3D textures, which are read as signed integer.
usampler3D	Sampler type for binding 3D textures, which are read as unsigned integer.
samplerCube	Sampler type for binding Cubemaps, which are read as float.
samplerCubeArray	Sampler type for binding Cubemap arrays, which are read as float. Only supported in Forward+ and Mobile, not Compatibility.

Comments

The shading language supports the same comment syntax as used in C# and C++:

```
// Single-line comment.
int a = 2;  // Another single-line comment.

/*
Multi-line comment.
The comment ends when the ending delimiter is found
(here, it's on the line below).
*/
int b = 3;
```

Additionally, you can use documentation comments that are displayed in the inspector when hovering a shader parameter. Documentation comments are currently only supported when placed immediately above a `uniform` declaration. These documentation comments only support the **multiline** comment syntax and must use **two** leading asterisks (`/**`) instead of just one (`/*`):

```
/**
 * This is a documentation comment.
 * These lines will appear in the inspector when hovering the
shader parameter
 * named "Something".
 * You can use [b]BBCode[/b] [i]formatting[/i] in the comment.
 */
uniform int something = 1;
```

The asterisks on the follow-up lines are not required, but are recommended as per the Shaders style guide. These asterisks are automatically stripped by the inspector, so they won't appear in the tooltip.

Casting

Just like GLSL ES 3.0, implicit casting between scalars and vectors of the same size but different type is not allowed. Casting of types of different size is also not allowed. Conversion must be done explicitly via constructors.

Example:

```
float a = 2; // invalid
float a = 2.0; // valid
float a = float(2); // valid
```

Default integer constants are signed, so casting is always needed to convert to unsigned:

```
int a = 2; // valid
uint a = 2; // invalid
uint a = uint(2); // valid
```

Members

Individual scalar members of vector types are accessed via the "x", "y", "z" and "w" members. Alternatively, using "r", "g", "b" and "a" also works and is equivalent. Use whatever fits best for your needs.

For matrices, use the `m[column][row]` indexing syntax to access each scalar, or `m[idx]` to access a vector by row index. For example, for accessing the y position of an object in a mat4 you use `m[3][1]`.

Constructing

Construction of vector types must always pass:

```
// The required amount of scalars
vec4 a = vec4(0.0, 1.0, 2.0, 3.0);
// Complementary vectors and/or scalars
vec4 a = vec4(vec2(0.0, 1.0), vec2(2.0, 3.0));
vec4 a = vec4(vec3(0.0, 1.0, 2.0), 3.0);
// A single scalar for the whole vector
vec4 a = vec4(0.0);
```

Construction of matrix types requires vectors of the same dimension as the matrix. You can also build a diagonal matrix using `matx(float)` syntax. Accordingly, `mat4(1.0)` is an identity matrix.

```
mat2 m2 = mat2(vec2(1.0, 0.0), vec2(0.0, 1.0));
mat3 m3 = mat3(vec3(1.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0),
vec3(0.0, 0.0, 1.0));
mat4 identity = mat4(1.0);
```

Matrices can also be built from a matrix of another dimension. There are two rules:

1. If a larger matrix is constructed from a smaller matrix, the additional rows and columns are set to the values they would have in an identity matrix. 2. If a smaller matrix is constructed from a larger matrix, the top, left submatrix of the larger matrix is used.

```
mat3 basis = mat3(MODEL_MATRIX);
mat4 m4 = mat4(basis);
mat2 m2 = mat2(m4);
```

Swizzling

It is possible to obtain any combination of components in any order, as long as the result is another vector type (or scalar). This is easier shown than explained:

```
vec4 a = vec4(0.0, 1.0, 2.0, 3.0);
vec3 b = a.rgb; // Creates a vec3 with vec4 components.
vec3 b = a.ggg; // Also valid; creates a vec3 and fills it with
a single vec4 component.
vec3 b = a.bgr; // "b" will be vec3(2.0, 1.0, 0.0).
vec3 b = a.xyz; // Also rgba, xyzw are equivalent.
vec3 b = a.stp; // And stpq (for texture coordinates).
float c = b.w; // Invalid, because "w" is not present in vec3
b.
vec3 c = b.xrt; // Invalid, mixing different styles is
forbidden.
b.rrr = a.rgb; // Invalid, assignment with duplication.
b.bgr = a.rgb; // Valid assignment. "b"'s "blue" component will
be "a"'s "red" and vice versa.
```

Precision

It is possible to add precision modifiers to datatypes; use them for uniforms, variables, arguments and varyings:

```
lowp vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // low precision,
usually 8 bits per component mapped to 0-1
mediump vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // medium precision,
usually 16 bits or half float
highp vec4 a = vec4(0.0, 1.0, 2.0, 3.0); // high precision,
uses full float or integer range (32 bit default)
```

Using lower precision for some operations can speed up the math involved (at the cost of less precision). This is rarely needed in the vertex processor function (where full precision is needed most of the time), but is often useful in the fragment processor.

Some architectures (mainly mobile) can benefit significantly from this, but there are downsides such as the additional overhead of conversion between precisions. Refer to the documentation of the target architecture for further information. In many cases, mobile drivers cause inconsistent or unexpected behavior and it is best to avoid specifying precision unless necessary.

Arrays

Arrays are containers for multiple variables of a similar type.

Local arrays

Local arrays are declared in functions. They can use all of the allowed datatypes, except samplers. The array declaration follows a C-style syntax: `[const] + [precision] + typename + identifier + [array size]`.

```
void fragment() {
    float arr[3];
}
```

They can be initialized at the beginning like:

```
float float_arr[3] = float[3] (1.0, 0.5, 0.0); // first
constructor

int int_arr[3] = int[] (2, 1, 0); // second constructor

vec2 vec2_arr[3] = { vec2(1.0, 1.0), vec2(0.5, 0.5), vec2(0.0,
0.0) }; // third constructor

bool bool_arr[] = { true, true, false }; // fourth constructor
- size is defined automatically from the element count
```

You can declare multiple arrays (even with different sizes) in one expression:

```
float a[3] = float[3] (1.0, 0.5, 0.0),
b[2] = { 1.0, 0.5 },
c[] = { 0.7 },
d = 0.0,
e[5];
```

To access an array element, use the indexing syntax:

```
float arr[3];

arr[0] = 1.0; // setter

COLOR.r = arr[0]; // getter
```

Arrays also have a built-in function `.length()` (not to be confused with the built-in `length()` function). It doesn't accept any parameters and will return the array's size.

```
float arr[] = { 0.0, 1.0, 0.5, -1.0 };
for (int i = 0; i < arr.length(); i++) {
    // ...
}
```

Note

If you use an index either below 0 or greater than array size - the shader will crash and break rendering. To prevent this, use `length()`, `if`, or `clamp()` functions to ensure the index is between 0 and the array's length. Always carefully test and check your code. If you pass a constant expression or a number, the editor will check its bounds to prevent this crash.

Global arrays

You can declare arrays at global space like:

```
shader_type spatial;

const lowp vec3 v[1] = lowp vec3[1] ( vec3(0, 0, 1) );

void fragment() {
  ALBEDO = v[0];
}
```

Note

Global arrays have to be declared as global constants, otherwise they can be declared the same as local arrays.

Constants

Use the `const` keyword before the variable declaration to make that variable immutable, which means that it cannot be modified. All basic types, except samplers can be declared as constants. Accessing and using a constant value is slightly faster than using a uniform. Constants must be initialized at their declaration.

```
const vec2 a = vec2(0.0, 1.0);
vec2 b;

a = b; // invalid
b = a; // valid
```

Constants cannot be modified and additionally cannot have hints, but multiple of them (if they have the same type) can be declared in a single expression e.g

```
const vec2 V1 = vec2(1, 1), V2 = vec2(2, 2);
```

Similar to variables, arrays can also be declared with `const`.

```
const float arr[] = { 1.0, 0.5, 0.0 };

arr[0] = 1.0; // invalid

COLOR.r = arr[0]; // valid
```

Constants can be declared both globally (outside of any function) or locally (inside a function). Global constants are useful when you want to have access to a value throughout your shader that does not need to be modified. Like uniforms, global constants are shared between all shader stages, but they are not accessible outside of the shader.

```
shader_type spatial;

const float PI = 3.14159265358979323846;
```

Constants of the `float` type must be initialized using `.` notation after the decimal part or by using the scientific notation. The optional `f` post-suffix is also supported.

```
float a = 1.0;
float b = 1.0f; // same, using suffix for clarity
float c = 1e-1; // gives 0.1 by using the scientific notation
```

Constants of the `uint` (unsigned int) type must have a `u` suffix to differentiate them from signed integers. Alternatively, this can be done by using the `uint(x)` built-in conversion function.

```
uint a = 1u;
uint b = uint(1);
```

Structs

Structs are compound types which can be used for better abstraction of shader code. You can declare them at the global scope like:

```
struct PointLight {
    vec3 position;
    vec3 color;
    float intensity;
};
```

After declaration, you can instantiate and initialize them like:

```
void fragment()
{
    PointLight light;
    light.position = vec3(0.0);
    light.color = vec3(1.0, 0.0, 0.0);
    light.intensity = 0.5;
}
```

Or use struct constructor for same purpose:

```
PointLight light = PointLight(vec3(0.0), vec3(1.0, 0.0, 0.0),
0.5);
```

Structs may contain other struct or array, you can also instance them as global constant:

```
shader_type spatial;

...
```

```
struct Scene {
    PointLight lights[2];
};

const Scene scene = Scene(PointLight[2](PointLight(vec3(0.0,
0.0, 0.0), vec3(1.0, 0.0, 0.0), 1.0), PointLight(vec3(0.0, 0.0,
0.0), vec3(1.0, 0.0, 0.0), 1.0)));

void fragment()
{
    ALBEDO = scene.lights[0].color;
}
```

You can also pass them to functions:

```
shader_type canvas_item;

...

Scene construct_scene(PointLight light1, PointLight light2) {
    return Scene({light1, light2});
}

void fragment()
{
    COLOR.rgb = construct_scene(PointLight(vec3(0.0, 0.0, 0.0),
vec3(1.0, 0.0, 0.0), 1.0), PointLight(vec3(0.0, 0.0, 0.0),
vec3(1.0, 0.0, 1.0), 1.0)).lights[0].color;
}
```

Operators

Godot shading language supports the same set of operators as GLSL ES 3.0. Below is the list of them in precedence order:

Precedence	Class	Operator

1 (highest)	parenthetical grouping	**()**
2	unary	**+, -, !, ~**
3	multiplicative	**/, *, %**
4	additive	**+, -**
5	bit-wise shift	**<<, >>**
6	relational	**<, >, <=, >=**
7	equality	**==, !=**
8	bit-wise AND	**&**
9	bit-wise exclusive OR	**^**
10	bit-wise inclusive OR	**|**
11	logical AND	**&&**
12 (lowest)	logical inclusive OR	**||**

Flow control

Godot Shading language supports the most common types of flow control:

```
// `if` and `else`.
if (cond) {

} else {

}

// Ternary operator.
// This is an expression that behaves like `if`/`else` and returns the value.
// If `cond` evaluates to `true`, `result` will be `9`.
// Otherwise, `result` will be `5`.
int result = cond ? 9 : 5;

// `switch`.
switch (i) { // `i` should be a signed integer expression.
    case -1:
        break;
    case 0:
        return; // `break` or `return` to avoid running the next `case`.
    case 1: // Fallthrough (no `break` or `return`): will run the next `case`.
    case 2:
        break;
    //...
    default: // Only run if no `case` above matches. Optional.
        break;
}

// `for` loop. Best used when the number of elements to iterate on
// is known in advance.
for (int i = 0; i < 10; i++) {

}
```

```
// `while` loop. Best used when the number of elements to
iterate on
// is not known in advance.
while (cond) {

}

// `do while`. Like `while`, but always runs at least once even
if `cond`
// never evaluates to `true`.
do {

} while (cond);
```

Keep in mind that in modern GPUs, an infinite loop can exist and can freeze your application (including editor). Godot can't protect you from this, so be careful not to make this mistake!

Also, when comparing floating-point values against a number, make sure to compare them against a *range* instead of an exact number.

A comparison like `if (value == 0.3)` may not evaluate to `true`. Floating-point math is often approximate and can defy expectations. It can also behave differently depending on the hardware.

Don't do this.

```
float value = 0.1 + 0.2;

// May not evaluate to `true`!
if (value == 0.3) {
    // ...
}
```

Instead, always perform a range comparison with an epsilon value. The larger the floating-point number (and the less precise the floating-point number), the larger the epsilon value should be.

```
const float EPSILON = 0.0001;
if (value >= 0.3 - EPSILON && value <= 0.3 + EPSILON) {
    // ...
}
```

See floating-point-gui.de [https://floating-point-gui.de/] for more information.

Discarding

Fragment and light functions can use the `discard` keyword. If used, the fragment is discarded and nothing is written.

Beware that `discard` has a performance cost when used, as it will prevent the depth prepass from being effective on any surfaces using the shader. Also, a discarded pixel still needs to be rendered in the vertex shader, which means a shader that uses `discard` on all of its pixels is still more expensive to render compared to not rendering any object in the first place.

Functions

It is possible to define functions in a Godot shader. They use the following syntax:

```
ret_type func_name(args) {
    return ret_type; // if returning a value
}

// a more specific example:

int sum2(int a, int b) {
    return a + b;
}
```

You can only use functions that have been defined above (higher in the editor) the function from which you are calling them. Redefining a function that has already been defined above (or is a built-in function name) will cause an error.

Function arguments can have special qualifiers:

- **in**: Means the argument is only for reading (default).
- **out**: Means the argument is only for writing.
- **inout**: Means the argument is fully passed via reference.
- **const**: Means the argument is a constant and cannot be changed, may be combined with **in** qualifier.

Example below:

```
void sum2(int a, int b, inout int result) {
    result = a + b;
}
```

Note

Unlike GLSL, Godot's shader language does **not** support function overloading. This means that a function cannot be defined several times with different argument types or numbers of arguments. As a workaround, use different names for functions that accept a different number of arguments or arguments of different types.

Varyings

To send data from the vertex to the fragment (or light) processor function, *varyings* are used. They are set for every primitive vertex in the *vertex processor*, and the value is interpolated for every pixel in the *fragment processor*.

```
shader_type spatial;

varying vec3 some_color;

void vertex() {
    some_color = NORMAL; // Make the normal the color.
}

void fragment() {
    ALBEDO = some_color;
}

void light() {
    DIFFUSE_LIGHT = some_color * 100; // optionally
}
```

Varying can also be an array:

```
shader_type spatial;

varying float var_arr[3];

void vertex() {
    var_arr[0] = 1.0;
    var_arr[1] = 0.0;
}

void fragment() {
    ALBEDO = vec3(var_arr[0], var_arr[1], var_arr[2]); // red
color
}
```

It's also possible to send data from *fragment* to *light* processors using *varying* keyword. To do so you can assign it in the *fragment* and later use it in the *light* function.

```
shader_type spatial;

varying vec3 some_light;

void fragment() {
    some_light = ALBEDO * 100.0; // Make a shining light.
}

void light() {
    DIFFUSE_LIGHT = some_light;
}
```

Note that varying may not be assigned in custom functions or a *light processor* function like:

```
shader_type spatial;

varying float test;

void foo() {
    test = 0.0; // Error.
}

void vertex() {
    test = 0.0;
}

void light() {
    test = 0.0; // Error too.
}
```

This limitation was introduced to prevent incorrect usage before initialization.

Interpolation qualifiers

Certain values are interpolated during the shading pipeline. You can modify how these interpolations are done by using *interpolation qualifiers*.

```
shader_type spatial;

varying flat vec3 our_color;

void vertex() {
    our_color = COLOR.rgb;
}

void fragment() {
    ALBEDO = our_color;
}
```

There are two possible interpolation qualifiers:

Qualifier	Description
flat	The value is not interpolated.
smooth	The value is interpolated in a perspective-correct fashion. This is the default.

Uniforms

Passing values to shaders is possible. These are global to the whole shader and are called *uniforms*. When a shader is later assigned to a material, the uniforms will appear as editable parameters in it. Uniforms can't be written from within the shader.

```
shader_type spatial;

uniform float some_value;

uniform vec3 colors[3];
```

You can set uniforms in the editor in the material. Or you can set them through GDScript:

```
material.set_shader_parameter("some_value", some_value)

material.set_shader_parameter("colors", [Vector3(1, 0, 0),
Vector3(0, 1, 0), Vector3(0, 0, 1)])
```

Note

The first argument to `set_shader_parameter` is the name of the uniform in the shader. It must match *exactly* to the name of the uniform in the shader or else it will not be recognized.

Any GLSL type except for *void* can be a uniform. Additionally, Godot provides optional shader hints to make the compiler understand for what the uniform is used, and how the editor should allow users to modify it.

```
shader_type spatial;

uniform vec4 color : source_color;
uniform float amount : hint_range(0, 1);
uniform vec4 other_color : source_color = vec4(1.0); // Default
values go after the hint.
uniform sampler2D image : source_color;
```

It's important to understand that textures *that are supplied as color* require hints for proper sRGB -> linear conversion (i.e. `source_color`), as Godot's 3D engine renders in linear color space. If this is not done, the texture will appear washed out.

Note

The 2D renderer also renders in linear color space if the **Rendering > Viewport > HDR 2D** project setting is enabled, so `source_color` must also be used in `canvas_item` shaders. If 2D HDR is disabled, `source_color` will keep working correctly in `canvas_item` shaders, so it's recommend to use it either way.

Full list of hints below:

Type	Hint	Description
vec3, vec4	source_color	Used as color.
int, float	hint_range(min, max[, step])	Restricted to values in a range (with min/max/step).
sampler2D	source_color	Used as albedo color.
sampler2D	hint_normal	Used as normalmap.
sampler2D	hint_default_white	As value or albedo color, default to opaque white.
sampler2D	hint_default_black	As value or albedo color, default to opaque black.
sampler2D	hint_default_transparent	As value or albedo color, default to transparent black.

Type	Hint	Description
sampler2D	hint_anisotropy	As flowmap, default to right.
sampler2D	hint_roughness[_r, _g, _b, _a, _normal, _gray]	Used for roughness limiter on import (attempts reducing specular aliasing). `_normal` is a normal map that guides the roughness limiter, with roughness increasing in areas that have high-frequency detail.
sampler2D	filter[_nearest, _linear][_mipmap][_anisotropic]	Enabled specified texture filtering.
sampler2D	repeat[_enable, _disable]	Enabled texture repeating.
sampler2D	hint_screen_texture	Texture is the screen texture.
sampler2D	hint_depth_texture	Texture is the depth texture.

Type	Hint	Description
sampler2D	hint_normal_roughness_texture	Texture is the normal roughness texture (only supported in Forward+).

GDScript uses different variable types than GLSL does, so when passing variables from GDScript to shaders, Godot converts the type automatically. Below is a table of the corresponding types:

GLSL type	GDScript type	Notes
bool	**bool**	
bvec2	**int**	Bitwise packed int where bit 0 (LSB) corresponds to x. For example, a bvec2 of (bx, by) could be created in the following way: `bvec2_input: int = (int(bx)) | (int(by) << 1)`
bvec3	**int**	Bitwise packed int where bit 0 (LSB) corresponds to x.

GLSL type	GDScript type	Notes
bvec4	**int**	Bitwise packed int where bit 0 (LSB) corresponds to x.
int	**int**	
ivec2	**Vector2i**	
ivec3	**Vector3i**	
ivec4	**Vector4i**	
uint	**int**	
uvec2	**Vector2i**	
uvec3	**Vector3i**	
uvec4	**Vector4i**	
float	**float**	
vec2	**Vector2**	

GLSL type	GDScript type	Notes
vec3	**Vector3**, **Color**	When Color is used, it will be interpreted as (r, g, b).
vec4	**Vector4**, **Color**, **Rect2**, **Plane**, **Quaternion**	When Color is used, it will be interpreted as (r, g, b, a). When Rect2 is used, it will be interpreted as (position.x, position.y, size.x, size.y). When Plane is used it will be interpreted as (normal.x, normal.y, normal.z, d).
mat2	**Transform2D**	
mat3	**Basis**	
mat4	**Projection**, **Transform3D**	When a Transform3D is used, the w Vector is set to the identity.
sampler2D	**Texture2D**	

GLSL type	GDScript type	Notes
isampler2D	**Texture2D**	
usampler2D	**Texture2D**	
sampler2DArray	**Texture2DArray**	
isampler2DArray	**Texture2DArray**	
usampler2DArray	**Texture2DArray**	
sampler3D	**Texture3D**	
isampler3D	**Texture3D**	
usampler3D	**Texture3D**	
samplerCube	**Cubemap**	See Changing import type for instructions on importing cubemaps for use in Godot.
samplerCubeArray	**CubemapArray**	Only supported in Forward+ and Mobile, not Compatibility.

Note

Be careful when setting shader uniforms from GDScript, no error will be thrown if the type does not match. Your shader will just exhibit undefined behavior.

Warning

As with the last note, no error will be thrown if the typing does not match while setting a shader uniform, this unintuitively includes setting a (GDscript) 64 bit int/float into a Godot shader language int/float (32 bit). This may lead to unintentional consequences in cases where high precision is required.

Uniforms can also be assigned default values:

```
shader_type spatial;

uniform vec4 some_vector = vec4(0.0);
uniform vec4 some_color : source_color = vec4(1.0);
```

Note that when adding a default value and a hint, the default value goes after the hint.

If you need to make multiple uniforms to be grouped in the specific category of an inspector, you can use a *group_uniform* keyword like:

```
group_uniforms MyGroup;
uniform sampler2D test;
```

You can close the group by using:

```
group_uniforms;
```

The syntax also supports subgroups (it's not mandatory to declare the base group before this):

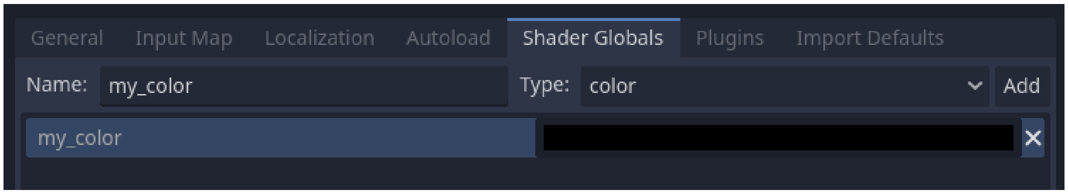
```
group_uniforms MyGroup.MySubgroup;
```

Global uniforms

Sometimes, you want to modify a parameter in many different shaders at once. With a regular uniform, this takes a lot of work as all these shaders need to be tracked and the uniform needs to be set for each of them. Global uniforms allow you to create and update uniforms that will be available in all shaders, in every shader type (`canvas_item`, `spatial`, `particles`, `sky` and `fog`).

Global uniforms are especially useful for environmental effects that affect many objects in a scene, like having foliage bend when the player is nearby, or having objects move with the wind.

To create a global uniform, open the **Project Settings** then go to the **Shader Globals** tab. Specify a name for the uniform (case-sensitive) and a type, then click **Add** in the top-right corner of the dialog. You can then edit the value assigned to the uniform by clicking the value in the list of uniforms:



Name: my_color

Type: color

my_color

Adding a global uniform in the Shader Globals tab of the Project Settings

After creating a global uniform, you can use it in a shader as follows:

```
shader_type canvas_item;

global uniform vec4 my_color;

void fragment() {
    COLOR = my_color.rgb;
}
```

Note that the global uniform *must* exist in the Project Settings at the time the shader is saved, or compilation will fail. While you can assign a default value using `global uniform vec4 my_color = ...` in the shader code, it will be ignored as the global uniform must always be defined in the Project Settings anyway.

To change the value of a global uniform at run-time, use the RenderingServer.global_shader_parameter_set method in a script:

```
RenderingServer.global_shader_parameter_set("my_color",
Color(0.3, 0.6, 1.0))
```

Assigning global uniform values can be done as many times as desired without impacting performance, as setting data doesn't require synchronization between the CPU and GPU.

You can also add or remove global uniforms at run-time:

```
RenderingServer.global_shader_parameter_add("my_color",
RenderingServer.GLOBAL_VAR_TYPE_COLOR, Color(0.3, 0.6, 1.0))
RenderingServer.global_shader_parameter_remove("my_color")
```

Adding or removing global uniforms at run-time has a performance cost, although it's not as pronounced compared to getting global uniform values from a script (see the warning below).

Warning

While you *can* query the value of a global uniform at run-time in a script using `RenderingServer.global_shader_parameter_get("uniform_name")`, this has a large performance penalty as the rendering thread needs to synchronize with the calling thread.

Therefore, it's not recommended to read global shader uniform values continuously in a script. If you need to read values in a script after setting them, consider creating an autoload where you store the values you need to query at the same time you're setting them as global uniforms.

Per-instance uniforms

Note

Per-instance uniforms are only available in `spatial` (3D) shaders.

Note

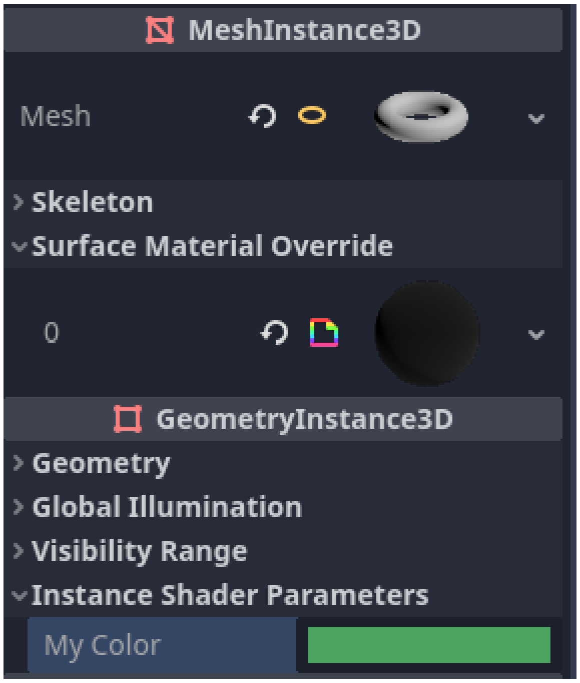
Per-instance uniforms are not supported when using the Compatibility renderer.

Sometimes, you want to modify a parameter on each node using the material. As an example, in a forest full of trees, when you want each tree to have a slightly different color that is editable by hand. Without per-instance uniforms, this requires creating a unique material for each tree (each with a slightly different hue). This makes material management more complex, and also has a performance overhead due to the scene requiring more unique material instances. Vertex colors could also be used here, but they'd require creating unique copies of the mesh for each different color, which also has a performance overhead.

Per-instance uniforms are set on each GeometryInstance3D, rather than on each Material instance. Take this into account when working with meshes that have multiple materials assigned to them, or MultiMesh setups.

```
shader_type spatial;

// Provide a hint to edit as a color. Optionally, a default
value can be provided.
// If no default value is provided, the type's default is used
(e.g. opaque black for colors).
instance uniform vec4 my_color : source_color = vec4(1.0, 0.5,
0.0, 1.0);

void fragment() {
    ALBEDO = my_color.rgb;
}
```

After saving the shader, you can change the per-instance uniform's value using the inspector:


MeshInstance3D
Mesh
Skeleton
Surface Material Override
0
GeometryInstance3D
Geometry
Global Illumination
Visibility Range
Instance Shader Parameters
My Color

Setting a per-instance uniform's value in the GeometryInstance3D section of the inspector

Per-instance uniform values can also be set at run-time using [set_instance_shader_parameter]() method on a node that inherits from [GeometryInstance3D]():

```
$MeshInstance3D.set_instance_shader_parameter("my_color",
Color(0.3, 0.6, 1.0))
```

When using per-instance uniforms, there are some restrictions you should be aware of:

- **Per-instance uniforms do not support textures**, only regular scalar and vector types. As a workaround, you can pass a texture array as a regular uniform, then pass the index of the texture to be drawn using a per-instance uniform.
- There is a practical maximum limit of 16 instance uniforms per shader.
- If your mesh uses multiple materials, the parameters for the first mesh material found will "win" over the subsequent ones, unless they have the same name, index *and* type. In this case, all parameters are affected correctly.
- If you run into the above situation, you can avoid clashes by manually specifying the index (0-15) of the instance uniform by using the `instance_index` hint:

```
instance uniform vec4 my_color : source_color,
instance_index(5);
```

Built-in variables

A large number of built-in variables are available, like `UV`, `COLOR` and `VERTEX`. What variables are available depends on the type of shader (`spatial`, `canvas_item` or `particle`) and the function used (`vertex`, `fragment` or `light`). For a list of the built-in variables that are available, please see the corresponding pages:

- Spatial shaders
- Canvas item shaders
- Particle shaders
- Sky shaders
- Fog shaders

Built-in functions

A large number of built-in functions are supported, conforming to GLSL ES 3.0. When vec_type (float), vec_int_type, vec_uint_type, vec_bool_type nomenclature is used, it can be scalar or vector.

Function	Description / Return value
vec_type **radians** (vec_type degrees)	Convert degrees to radians.
vec_type **degrees** (vec_type radians)	Convert radians to degrees.
vec_type **sin** (vec_type x)	Sine.
vec_type **cos** (vec_type x)	Cosine.
vec_type **tan** (vec_type x)	Tangent.
vec_type **asin** (vec_type x)	Arcsine.
vec_type **acos** (vec_type x)	Arccosine.
vec_type **atan** (vec_type y_over_x)	Arctangent.

Function	**Description / Return value**
vec_type **atan** (vec_type y, vec_type x)	Arctangent.
vec_type **sinh** (vec_type x)	Hyperbolic sine.
vec_type **cosh** (vec_type x)	Hyperbolic cosine.
vec_type **tanh** (vec_type x)	Hyperbolic tangent.
vec_type **asinh** (vec_type x)	Inverse hyperbolic sine.
vec_type **acosh** (vec_type x)	Inverse hyperbolic cosine.
vec_type **atanh** (vec_type x)	Inverse hyperbolic tangent.
vec_type **pow** (vec_type x, vec_type y)	Power (undefined if `x` < 0 or if `x` == 0 and `y` <= 0).
vec_type **exp** (vec_type x)	Base-e exponential.
vec_type **exp2** (vec_type x)	Base-2 exponential.
vec_type **log** (vec_type x)	Natural logarithm.

Function	**Description / Return value**
vec_type **log2** (vec_type x)	Base-2 logarithm.
vec_type **sqrt** (vec_type x)	Square root.
vec_type **inversesqrt** (vec_type x)	Inverse square root.
vec_type **abs** (vec_type x) ivec_type **abs** (ivec_type x)	Absolute value (returns positive value if negative).
vec_type **sign** (vec_type x) ivec_type **sign** (ivec_type x)	Sign (returns `1.0` if positive, `-1.0` if negative, `0.0` if zero).
vec_type **floor** (vec_type x)	Round to the integer below.
vec_type **round** (vec_type x)	Round to the nearest integer.
vec_type **roundEven** (vec_type x)	Round to the nearest even integer.
vec_type **trunc** (vec_type x)	Truncation.
vec_type **ceil** (vec_type x)	Round to the integer above.

Function	Description / Return value
vec_type **fract** (vec_type x)	Fractional (returns `x - floor(x)`).
vec_type **mod** (vec_type x, vec_type y) vec_type **mod** (vec_type x, float y)	Modulo (division remainder).
vec_type **modf** (vec_type x, out vec_type i)	Fractional of `x`, with `i` as integer part.
vec_type **min** (vec_type a, vec_type b)	Lowest value between `a` and `b`.
vec_type **max** (vec_type a, vec_type b)	Highest value between `a` and `b`.
vec_type **clamp** (vec_type x, vec_type min, vec_type max)	Clamp `x` between `min` and `max` (inclusive).
float **mix** (float a, float b, float c) vec_type **mix** (vec_type a, vec_type b, float c)	Linear interpolate between `a` and `b` by `c`.

Function	Description / Return value
vec_type **mix** (vec_type a, vec_type b, bvec_type c)	
vec_type **fma** (vec_type a, vec_type b, vec_type c)	Performs a fused multiply-add operation: `(a * b + c)` (faster than doing it manually).
vec_type **step** (vec_type a, vec_type b)	`b[i] < a[i] ? 0.0 : 1.0`.
vec_type **step** (float a, vec_type b)	`b[i] < a ? 0.0 : 1.0`.
vec_type **smoothstep** (vec_type a, vec_type b, vec_type c) vec_type **smoothstep** (float a, float b, vec_type c)	Hermite interpolate between `a` and `b` by `c`.
bvec_type **isnan** (vec_type x)	Returns `true` if scalar or vector component is `NaN`.
bvec_type **isinf** (vec_type x)	Returns `true` if scalar or vector component is `INF`.
ivec_type **floatBitsToInt** (vec_type x)	Float->Int bit copying, no conversion.

Function	Description / Return value
uvec_type **floatBitsToUint** (vec_type x)	Float->UInt bit copying, no conversion.
vec_type **intBitsToFloat** (ivec_type x)	Int->Float bit copying, no conversion.
vec_type **uintBitsToFloat** (uvec_type x)	UInt->Float bit copying, no conversion.
float **length** (vec_type x)	Vector length.
float **distance** (vec_type a, vec_type b)	Distance between vectors i.e `length(a - b)`.
float **dot** (vec_type a, vec_type b)	Dot product.
vec3 **cross** (vec3 a, vec3 b)	Cross product.
vec_type **normalize** (vec_type x)	Normalize to unit length.
vec3 **reflect** (vec3 I, vec3 N)	Reflect.
vec3 **refract** (vec3 I, vec3 N, float eta)	Refract.

Function	**Description / Return value**
vec_type **faceforward** (vec_type N, vec_type I, vec_type Nref)	If `dot(Nref, I)` < 0, return `N`, otherwise `-N`.
mat_type **matrixCompMult** (mat_type x, mat_type y)	Matrix component multiplication.
mat_type **outerProduct** (vec_type column, vec_type row)	Matrix outer product.
mat_type **transpose** (mat_type m)	Transpose matrix.
float **determinant** (mat_type m)	Matrix determinant.
mat_type **inverse** (mat_type m)	Inverse matrix.
bvec_type **lessThan** (vec_type x, vec_type y)	Bool vector comparison on < int/uint/float vectors.
bvec_type **greaterThan** (vec_type x, vec_type y)	Bool vector comparison on > int/uint/float vectors.
bvec_type **lessThanEqual** (vec_type x, vec_type y)	Bool vector comparison on <= int/uint/float vectors.

Function	**Description / Return value**
bvec_type **greaterThanEqual** (vec_type x, vec_type y)	Bool vector comparison on >= int/uint/float vectors.
bvec_type **equal** (vec_type x, vec_type y)	Bool vector comparison on == int/uint/float vectors.
bvec_type **notEqual** (vec_type x, vec_type y)	Bool vector comparison on != int/uint/float vectors.
bool **any** (bvec_type x)	`true` if any component is `true`, `false` otherwise.
bool **all** (bvec_type x)	`true` if all components are `true`, `false` otherwise.
bvec_type **not** (bvec_type x)	Invert boolean vector.
ivec2 **textureSize** (gsampler2D s, int lod) ivec3 **textureSize** (gsampler2DArray s, int lod) ivec3 **textureSize** (gsampler3D s, int lod)	Get the size of a texture. The LOD defines which mipmap level is used. An LOD value of `0` will use the full resolution texture.

Function	Description / Return value
ivec2 **textureSize** (samplerCube s, int lod) ivec2 **textureSize** (samplerCubeArray s, int lod)	
vec2 **textureQueryLod** (gsampler2D s, vec2 p) vec3 **textureQueryLod** (gsampler2DArray s, vec2 p) vec2 **textureQueryLod** (gsampler3D s, vec3 p) vec2 **textureQueryLod** (samplerCube s, vec3 p)	Compute the level-of-detail that would be used to sample from a texture. The `x` component of the resulted value is the mipmap array that would be accessed. The `y` component is computed level-of-detail relative to the base level (regardless of the mipmap levels of the texture).
int **textureQueryLevels** (gsampler2D s) int **textureQueryLevels** (gsampler2DArray s) int **textureQueryLevels** (gsampler3D s) int **textureQueryLevels** (samplerCube s)	Get the number of accessible mipmap levels of a texture. If the texture is unassigned to a sampler, `1` is returned (Godot always internally assigns a texture even to an empty sampler).
gvec4_type **texture** (gsampler2D s, vec2 p [, float bias])	Perform a texture read.

Function	Description / Return value
gvec4_type **texture** (gsampler2DArray s, vec3 p [, float bias]) gvec4_type **texture** (gsampler3D s, vec3 p [, float bias]) vec4 **texture** (samplerCube s, vec3 p [, float bias]) vec4 **texture** (samplerCubeArray s, vec4 p [, float bias])	
gvec4_type **textureProj** (gsampler2D s, vec3 p [, float bias]) gvec4_type **textureProj** (gsampler2D s, vec4 p [, float bias]) gvec4_type **textureProj** (gsampler3D s, vec4 p [, float bias])	Perform a texture read with projection.
gvec4_type **textureLod** (gsampler2D s, vec2 p, float lod) gvec4_type **textureLod** (gsampler2DArray s, vec3 p, float lod)	Perform a texture read at custom mipmap. The LOD defines which mipmap level is used. An LOD value of `0.0` will use the full

Function	Description / Return value
gvec4_type **textureLod** (gsampler3D s, vec3 p, float lod) vec4 **textureLod** (samplerCube s, vec3 p, float lod) vec4 **textureLod** (samplerCubeArray s, vec4 p, float lod)	resolution texture. If the texture lacks mipmaps, all LOD values will act like `0.0`.
gvec4_type **textureProjLod** (gsampler2D s, vec3 p, float lod) gvec4_type **textureProjLod** (gsampler2D s, vec4 p, float lod) gvec4_type **textureProjLod** (gsampler3D s, vec4 p, float lod)	Performs a texture read with projection/LOD. The LOD defines which mipmap level is used. An LOD value of `0.0` will use the full resolution texture. If the texture lacks mipmaps, all LOD values will act like `0.0`.
gvec4_type **textureGrad** (gsampler2D s, vec2 p, vec2 dPdx, vec2 dPdy) gvec4_type **textureGrad** (gsampler2DArray s, vec3 p, vec2 dPdx, vec2 dPdy) gvec4_type **textureGrad** (gsampler3D s, vec3 p, vec2 dPdx, vec2 dPdy)	Performs a texture read with explicit gradients.

Function	Description / Return value
vec4 **textureGrad** (samplerCube s, vec3 p, vec3 dPdx, vec3 dPdy) vec4 **textureGrad** (samplerCubeArray s, vec3 p, vec3 dPdx, vec3 dPdy)	
gvec4_type **textureProjGrad** (gsampler2D s, vec3 p, vec2 dPdx, vec2 dPdy) gvec4_type **textureProjGrad** (gsampler2D s, vec4 p, vec2 dPdx, vec2 dPdy) gvec4_type **textureProjGrad** (gsampler3D s, vec4 p, vec3 dPdx, vec3 dPdy)	Performs a texture read with projection/LOD and with explicit gradients.
gvec4_type **texelFetch** (gsampler2D s, ivec2 p, int lod) gvec4_type **texelFetch** (gsampler2DArray s, ivec3 p, int lod) gvec4_type **texelFetch** (gsampler3D s, ivec3 p, int lod)	Fetches a single texel using integer coordinates. The LOD defines which mipmap level is used. An LOD value of `0` will use the full resolution texture.
gvec4_type **textureGather** (gsampler2D s, vec2 p [, int	Gathers four texels from a texture. Use `comps` within

Function	**Description / Return value**
comps]) gvec4_type **textureGather** (gsampler2DArray s, vec3 p [, int comps]) vec4 **textureGather** (samplerCube s, vec3 p [, int comps])	range of 0..3 to define which component (x, y, z, w) is returned. If `comps` is not provided: 0 (or x-component) is used.
vec_type **dFdx** (vec_type p)	Derivative in `x` using local differencing. Internally, can use either `dFdxCoarse` or `dFdxFine`, but the decision for which to use is made by the GPU driver.
vec_type **dFdxCoarse** (vec_type p)	Calculates derivative with respect to `x` window coordinate using local differencing based on the value of `p` for the current fragment neighbour(s), and will possibly, but not necessarily, include the value for the current fragment. This function is not available on `gl_compatibility` profile.

Function	Description / Return value
vec_type **dFdxFine** (vec_type p)	Calculates derivative with respect to `x` window coordinate using local differencing based on the value of `p` for the current fragment and its immediate neighbour(s). This function is not available on `gl_compatibility` profile.
vec_type **dFdy** (vec_type p)	Derivative in `y` using local differencing. Internally, can use either `dFdyCoarse` or `dFdyFine`, but the decision for which to use is made by the GPU driver.
vec_type **dFdyCoarse** (vec_type p)	Calculates derivative with respect to `y` window coordinate using local differencing based on the value of `p` for the current fragment neighbour(s), and will possibly, but not necessarily, include the value for the current fragment. This function is not available on `gl_compatibility` profile.

Function	Description / Return value
vec_type **dFdyFine** (vec_type p)	Calculates derivative with respect to `y` window coordinate using local differencing based on the value of `p` for the current fragment and its immediate neighbour(s). This function is not available on `gl_compatibility` profile.
vec_type **fwidth** (vec_type p)	Sum of absolute derivative in `x` and `y`. This is the equivalent of using `abs(dFdx(p)) + abs(dFdy(p))`.
vec_type **fwidthCoarse** (vec_type p)	Sum of absolute derivative in `x` and `y`. This is the equivalent of using `abs(dFdxCoarse(p)) + abs(dFdyCoarse(p))`. This function is not available on `gl_compatibility` profile.
vec_type **fwidthFine** (vec_type p)	Sum of absolute derivative in `x` and `y`. This is the equivalent of using `abs(dFdxFine(p)) + abs(dFdyFine(p))`. This function is not available on `gl_compatibility` profile.

Function	**Description / Return value**
uint **packHalf2x16** (vec2 v) vec2 **unpackHalf2x16** (uint v)	Convert two 32-bit floating-point numbers into 16-bit and pack them into a 32-bit unsigned integer and vice-versa.
uint **packUnorm2x16** (vec2 v) vec2 **unpackUnorm2x16** (uint v)	Convert two 32-bit floating-point numbers (clamped within 0..1 range) into 16-bit and pack them into a 32-bit unsigned integer and vice-versa.
uint **packSnorm2x16** (vec2 v) vec2 **unpackSnorm2x16** (uint v)	Convert two 32-bit floating-point numbers (clamped within -1..1 range) into 16-bit and pack them into a 32-bit unsigned integer and vice-versa.
uint **packUnorm4x8** (vec4 v) vec4 **unpackUnorm4x8** (uint v)	Convert four 32-bit floating-point numbers (clamped within 0..1 range) into 8-bit and pack them into a 32-bit unsigned integer and vice-versa.
uint **packSnorm4x8** (vec4 v)	Convert four 32-bit floating-point numbers (clamped

Function	Description / Return value
vec4 **unpackSnorm4x8** (uint v)	within -1..1 range) into 8-bit and pack them into a 32-bit unsigned integer and vice-versa.
ivec_type **bitfieldExtract** (ivec_type value, int offset, int bits) uvec_type **bitfieldExtract** (uvec_type value, int offset, int bits)	Extracts a range of bits from an integer.
ivec_type **bitfieldInsert** (ivec_type base, ivec_type insert, int offset, int bits) uvec_type **bitfieldInsert** (uvec_type base, uvec_type insert, int offset, int bits)	Insert a range of bits into an integer.
ivec_type **bitfieldReverse** (ivec_type value) uvec_type **bitfieldReverse** (uvec_type value)	Reverse the order of bits in an integer.
ivec_type **bitCount** (ivec_type value)	Counts the number of 1 bits in an integer.

Function	**Description / Return value**
uvec_type **bitCount** (uvec_type value)	
ivec_type **findLSB** (ivec_type value) uvec_type **findLSB** (uvec_type value)	Find the index of the least significant bit set to 1 in an integer.
ivec_type **findMSB** (ivec_type value) uvec_type **findMSB** (uvec_type value)	Find the index of the most significant bit set to 1 in an integer.
void **imulExtended** (ivec_type x, ivec_type y, out ivec_type msb, out ivec_type lsb) void **umulExtended** (uvec_type x, uvec_type y, out uvec_type msb, out uvec_type lsb)	Multiplies two 32-bit numbers and produce a 64-bit result. `x` - the first number. `y` - the second number. `msb` - will contain the most significant bits. `lsb` - will contain the least significant bits.
uvec_type **uaddCarry** (uvec_type x, uvec_type y, out uvec_type carry)	Adds two unsigned integers and generates carry.
uvec_type **usubBorrow** (uvec_type x, uvec_type y, out	Subtracts two unsigned integers and generates

Function	**Description / Return value**
uvec_type borrow)	borrow.
vec_type **ldexp** (vec_type x, out ivec_type exp)	Assemble a floating-point number from a value and exponent. If this product is too large to be represented in the floating-point type the result is undefined.
vec_type **frexp** (vec_type x, out ivec_type exp)	Splits a floating-point number(`x`) into significand (in the range of [0.5, 1.0]) and an integral exponent. For `x` equals zero the significand and exponent are both zero. For `x` of infinity or NaN, the results are undefined.

Shader preprocessor

Why use a shader preprocessor?

In programming languages, a *preprocessor* allows changing the code before the compiler reads it. Unlike the compiler, the preprocessor does not care about whether the syntax of the preprocessed code is valid. The preprocessor always performs what the *directives* tell it to do. A directive is a statement starting with a hash symbol (`#`). It is not a *keyword* of the shader language (such as `if` or `for`), but a special kind of token within the language.

From Godot 4.0 onwards, you can use a shader preprocessor within text-based shaders. The syntax is similar to what most GLSL shader compilers support (which in turn is similar to the C/C++ preprocessor).

Note

The shader preprocessor is not available in visual shaders. If you need to introduce preprocessor statements to a visual shader, you can convert it to a text-based shader using the **Convert to Shader** option in the VisualShader inspector resource dropdown. This conversion is a one-way operation; text shaders cannot be converted back to visual shaders.

Directives

General syntax

- Preprocessor directives do not use brackets (`{}`), but can use parentheses.
- Preprocessor directives **never** end with semicolons (with the exception of `#define`, where this is allowed but potentially dangerous).
- Preprocessor directives can span several lines by ending each line with a backslash (`\`). The first line break *not* featuring a backslash will end the preprocessor statement.

#define

Syntax: `#define <identifier> [replacement_code]`.

Defines the identifier after that directive as a macro, and replaces all successive occurrences of it with the replacement code given in the shader. Replacement is performed on a "whole words" basis, which means no replacement is performed if the string is part of another string (without any spaces or operators separating it).

Defines with replacements may also have one or more *arguments*, which can then be passed when referencing the define (similar to a function call).

If the replacement code is not defined, the identifier may only be used with `#ifdef` or `#ifndef` directives.

If the *concatenation* symbol (`##`) is present in the replacement code then it will be removed upon macro insertion, together with any space surrounding it, and join the surrounding words and arguments into a new token.

```
uniform sampler2D material0;

#define SAMPLE(N) vec4 tex##N = texture(material##N, UV)

void fragment() {
    SAMPLE(0);
    ALBEDO = tex0.rgb;
}
```

Compared to constants (`const CONSTANT = value;`), `#define` can be used anywhere within the shader (including in uniform hints). `#define` can also be used to insert arbitrary shader code at any location, while constants can't do that.

```
shader_type spatial;

// Notice the lack of semicolon at the end of the line, as the
replacement text
// shouldn't insert a semicolon on its own.
// If the directive ends with a semicolon, the semicolon is
inserted in every usage
// of the directive, even when this causes a syntax error.
#define USE_MY_COLOR
#define MY_COLOR vec3(1, 0, 0)

// Replacement with arguments.
// All arguments are required (no default values can be
provided).
#define BRIGHTEN_COLOR(r, g, b) vec3(r + 0.5, g + 0.5, b + 0.5)

// Multiline replacement using backslashes for continuation:
#define SAMPLE(param1, param2, param3, param4)
long_function_call( \
        param1, \
        param2, \
        param3, \
        param4 \
)

void fragment() {
#ifdef USE_MY_COLOR
    ALBEDO = MY_COLOR;
#endif
}
```

Defining a `#define` for an identifier that is already defined results in an error. To prevent this, use `#undef <identifier>`.

#undef

Syntax: `#undef identifier`

The `#undef` directive may be used to cancel a previously defined `#define` directive:

```
#define MY_COLOR vec3(1, 0, 0)

vec3 get_red_color() {
    return MY_COLOR;
}

#undef MY_COLOR
#define MY_COLOR vec3(0, 1, 0)

vec3 get_green_color() {
    return MY_COLOR;
}

// Like in most preprocessors, undefining a define that was not
previously defined is allowed
// (and won't print any warning or error).
#undef THIS_DOES_NOT_EXIST
```

Without `#undef` in the above example, there would be a macro redefinition error.

#if

Syntax: `#if <condition>`

The `#if` directive checks whether the `condition` passed. If it evaluates to a non-zero value, the code block is included, otherwise it is skipped.

To evaluate correctly, the condition must be an expression giving a simple floating-point, integer or boolean result. There may be multiple condition blocks connected by `&&` (AND) or `||` (OR) operators. It may be continued by a `#else` block, but **must** be ended with the `#endif` directive.

```
#define VAR 3
#define USE_LIGHT 0 // Evaluates to `false`.
#define USE_COLOR 1 // Evaluates to `true`.

#if VAR == 3 && (USE_LIGHT || USE_COLOR)
// Condition is `true`. Include this portion in the final
shader.
#endif
```

Using the `defined()` *preprocessor function*, you can check whether the passed identifier is defined a by `#define` placed above that directive. This is useful for creating multiple shader versions in the same file. It may be continued by a `#else` block, but must be ended with the `#endif` directive.

The `defined()` function's result can be negated by using the `!` (boolean NOT) symbol in front of it. This can be used to check whether a define is *not* set.

```
#define USE_LIGHT
#define USE_COLOR

// Correct syntax:
#if defined(USE_LIGHT) || defined(USE_COLOR) ||
!defined(USE_REFRACTION)
// Condition is `true`. Include this portion in the final
shader.
#endif
```

Be careful, as `defined()` must only wrap a single identifier within parentheses, never more:

```
// Incorrect syntax (parentheses are not placed where they
should be):
#if defined(USE_LIGHT || USE_COLOR || !USE_REFRACTION)
// This will cause an error or not behave as expected.
#endif
```

Tip

In the shader editor, preprocessor branches that evaluate to `false` (and are therefore excluded from the final compiled shader) will appear grayed out. This does not apply to run-time `if` statements.

#if preprocessor versus if statement: Performance caveats

The shading language supports run-time `if` statements:

```
uniform bool USE_LIGHT = true;

if (USE_LIGHT) {
    // This part is included in the compiled shader, and always
run.
} else {
    // This part is included in the compiled shader, but never
run.
}
```

If the uniform is never changed, this behaves identical to the following usage of the `#if` preprocessor statement:

```
#define USE_LIGHT

#if defined(USE_LIGHT)
// This part is included in the compiled shader, and always
run.
#else
// This part is *not* included in the compiled shader (and
therefore never run).
#endif
```

However, the `#if` variant can be faster in certain scenarios. This is because all run-time branches in a shader are still compiled and variables within those branches may still take up register space, even if they are never run in practice.

Modern GPUs are quite effective [https://medium.com/@jasonbooth_86226/branching-on-a-gpu-18bfc83694f2] at performing "static" branching. "Static" branching refers to `if` statements where *all* pixels/vertices evaluate to the same result in a given shader invocation. However, high amounts of VGPRs (which can be caused by having too many branches) can still slow down shader execution significantly.

#elif

The `#elif` directive stands for "else if" and checks the condition passed if the above `#if` evaluated to `false`. `#elif` can only be used within an `#if` block. It is possible to use several `#elif` statements after an `#if` statement.

```
#define VAR 2

#if VAR == 0
// Not included.
#elif VAR == 1
// Not included.
#elif VAR == 2
// Condition is `true`. Include this portion in the final
shader.
#else
// Not included.
#endif
```

Like with `#if`, the `defined()` preprocessor function can be used:

```
#define SHADOW_QUALITY_MEDIUM

#if defined(SHADOW_QUALITY_HIGH)
```

```
// High shadow quality.
#elif defined(SHADOW_QUALITY_MEDIUM)
// Medium shadow quality.
#else
// Low shadow quality.
#endif
```

#ifdef

Syntax: `#ifdef <identifier>`

This is a shorthand for `#if defined(...)`. Checks whether the passed identifier is defined by `#define` placed above that directive. This is useful for creating multiple shader versions in the same file. It may be continued by a `#else` block, but must be ended with the `#endif` directive.

```
#define USE_LIGHT

#ifdef USE_LIGHT
// USE_LIGHT is defined. Include this portion in the final
shader.
#endif
```

The processor does *not* support `#elifdef` as a shortcut for `#elif defined(...)`. Instead, use the following series of `#ifdef` and `#else` when you need more than two branches:

```
#define SHADOW_QUALITY_MEDIUM

#ifdef SHADOW_QUALITY_HIGH
// High shadow quality.
#else
#ifdef SHADOW_QUALITY_MEDIUM
// Medium shadow quality.
#else
// Low shadow quality.
```

```
#endif // This ends `SHADOW_QUALITY_MEDIUM`'s branch.
#endif // This ends `SHADOW_QUALITY_HIGH`'s branch.
```

#ifndef

Syntax: `#ifndef <identifier>`

This is a shorthand for `#if !defined(...)`. Similar to `#ifdef`, but checks whether the passed identifier is **not** defined by `#define` before that directive.

This is the exact opposite of `#ifdef`; it will always match in situations where `#ifdef` would never match, and vice versa.

```
#define USE_LIGHT

#ifndef USE_LIGHT
// Evaluates to `false`. This portion won't be included in the
final shader.
#endif

#ifndef USE_COLOR
// Evaluates to `true`. This portion will be included in the
final shader.
#endif
```

#else

Syntax: `#else`

Defines the optional block which is included when the previously defined `#if`, `#elif`, `#ifdef` or `#ifndef` directive evaluates to false.

```
shader_type spatial;

#define MY_COLOR vec3(1.0, 0, 0)
```

```
void fragment() {
#ifdef MY_COLOR
    ALBEDO = MY_COLOR;
#else
    ALBEDO = vec3(0, 0, 1.0);
#endif
}
```

#endif

Syntax: `#endif`

Used as terminator for the `#if`, `#ifdef`, `#ifndef` or subsequent `#else` directives.

#include

Syntax: `#include "path"`

The `#include` directive includes the *entire* content of a shader include file in a shader. `"path"` can be an absolute `res://` path or relative to the current shader file. Relative paths are only allowed in shaders that are saved to `.gdshader` or `.gdshaderinc` files, while absolute paths can be used in shaders that are built into a scene/resource file.

You can create new shader includes by using the **File > Create Shader Include** menu option of the shader editor, or by creating a new ShaderInclude resource in the FileSystem dock.

Shader includes can be included from within any shader, or other shader include, at any point in the file.

When including shader includes in the global scope of a shader, it is recommended to do this after the initial `shader_type` statement.

You can also include shader includes from within the body a function. Please note that the shader editor is likely going to report errors for your shader include's code, as it may not be valid outside of the context that it was written for. You can either choose to ignore these errors (the shader will still compile fine), or you can wrap the include in an `#ifdef` block that checks for a define from your shader.

`#include` is useful for creating libraries of helper functions (or macros) and reducing code duplication. When using `#include`, be careful about naming collisions, as redefining functions or macros is not allowed.

`#include` is subject to several restrictions:

- Only shader include resources (ending with `.gdshaderinc`) can be included. `.gdshader` files cannot be included by another shader, but a `.gdshaderinc` file can include other `.gdshaderinc` files.
- Cyclic dependencies are **not** allowed and will result in an error.
- To avoid infinite recursion, include depth is limited to 25 steps.

Example shader include file:

```
// fancy_color.gdshaderinc

// While technically allowed, there is usually no `shader_type` declaration in include files.

vec3 get_fancy_color() {
    return vec3(0.3, 0.6, 0.9);
}
```

Example base shader (using the include file we created above):

```
// material.gdshader

shader_type spatial;

#include "res://fancy_color.gdshaderinc"
```

```
void fragment() {
    // No error, as we've included a definition for
`get_fancy_color()` via the shader include.
    COLOR = get_fancy_color();
}
```

#pragma

Syntax: `#pragma value`

The `#pragma` directive provides additional information to the preprocessor or compiler.

Currently, it may have only one value: `disable_preprocessor`. If you don't need the preprocessor, use that directive to speed up shader compilation by excluding the preprocessor step.

```
#pragma disable_preprocessor

#if USE_LIGHT
// This causes a shader compilation error, as the `#if
USE_LIGHT` and `#endif`
// are included as-is in the final shader code.
#endif
```

Spatial shaders

Spatial shaders are used for shading 3D objects. They are the most complex type of shader Godot offers. Spatial shaders are highly configurable with different render modes and different rendering options (e.g. Subsurface Scattering, Transmission, Ambient Occlusion, Rim lighting etc). Users can optionally write vertex, fragment, and light processor functions to affect how objects are drawn.

Render modes

Render mode	Description
blend_mix	Mix blend mode (alpha is transparency), default.
blend_add	Additive blend mode.
blend_sub	Subtractive blend mode.
blend_mul	Multiplicative blend mode.
depth_draw_opaque	Only draw depth for opaque geometry (not transparent).

Render mode	Description
depth_draw_always	Always draw depth (opaque and transparent).
depth_draw_never	Never draw depth.
depth_prepass_alpha	Do opaque depth pre-pass for transparent geometry.
depth_test_disabled	Disable depth testing.
sss_mode_skin	Subsurface Scattering mode for skin.
cull_back	Cull back-faces (default).
cull_front	Cull front-faces.
cull_disabled	Culling disabled (double sided).
unshaded	Result is just albedo. No lighting/shading happens in material.
wireframe	Geometry draws using lines.

Render mode	Description
diffuse_burley	Burley (Disney PBS) for diffuse (default).
diffuse_lambert	Lambert shading for diffuse.
diffuse_lambert_wrap	Lambert wrapping (roughness dependent) for diffuse.
diffuse_toon	Toon shading for diffuse.
specular_schlick_ggx	Schlick-GGX for specular (default).
specular_toon	Toon for specular.
specular_disabled	Disable specular.
skip_vertex_transform	VERTEX/NORMAL/etc. need to be transformed manually in vertex function.
world_vertex_coords	VERTEX/NORMAL/etc. are modified in world coordinates instead of local.

Render mode	Description
ensure_correct_normals	Use when non-uniform scale is applied to mesh.
shadows_disabled	Disable computing shadows in shader.
ambient_light_disabled	Disable contribution from ambient light and radiance map.
shadow_to_opacity	Lighting modifies the alpha so shadowed areas are opaque and non-shadowed areas are transparent. Useful for overlaying shadows onto a camera feed in AR.
vertex_lighting	Use vertex-based lighting.
particle_trails	Enables the trails when used on particles geometry.
alpha_to_coverage	Alpha antialiasing mode, see here for more.
alpha_to_coverage_and_one	Alpha antialiasing mode, see here for more.

Render mode	Description
fog_disabled	Disable receiving depth-based or volumetric fog. Useful for blend_add materials like particles.

Built-ins

Values marked as "in" are read-only. Values marked as "out" are for optional writing and will not necessarily contain sensible values. Values marked as "inout" provide a sensible default value, and can optionally be written to. Samplers are not subjects of writing and they are not marked.

Global built-ins

Global built-ins are available everywhere, including custom functions.

Built-in	Description
in float **TIME**	Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's not affected by time_scale or pausing. If you need a `TIME` variable that can be scaled or paused, add your own global shader uniform and update it each frame.

Built-in	Description
in float **PI**	A `PI` constant (`3.141592`). A ration of circle's circumference to its diameter and amount of radians in half turn.
in float **TAU**	A `TAU` constant (`6.283185`). An equivalent of `PI * 2` and amount of radians in full turn.
in float **E**	An `E` constant (`2.718281`). Euler's number and a base of the natural logarithm.

Vertex built-ins

Vertex data (`VERTEX`, `NORMAL`, `TANGENT`, `BITANGENT`) are presented in local model space. If not written to, these values will not be modified and be passed through as they came.

They can optionally be presented in world space by using the *world_vertex_coords* render mode.

Users can disable the built-in modelview transform (projection will still happen later) and do it manually with the following code:

```
shader_type spatial;
render_mode skip_vertex_transform;

void vertex() {
    VERTEX = (MODELVIEW_MATRIX * vec4(VERTEX, 1.0)).xyz;
    NORMAL = normalize((MODELVIEW_MATRIX * vec4(NORMAL,
0.0)).xyz);
    BINORMAL = normalize((MODELVIEW_MATRIX * vec4(BINORMAL,
0.0)).xyz);
    TANGENT = normalize((MODELVIEW_MATRIX * vec4(TANGENT,
0.0)).xyz);
}
```

Other built-ins, such as UV, UV2 and COLOR, are also passed through to the fragment function if not modified.

Users can override the modelview and projection transforms using the `POSITION` built-in. If `POSITION` is written to anywhere in the shader, it will always be used, so the user becomes responsible for ensuring that it always has an acceptable value. When `POSITION` is used, the value from `VERTEX` is ignored and projection does not happen. However, the value passed to the fragment shader still comes from `VERTEX`.

For instancing, the INSTANCE_CUSTOM variable contains the instance custom data. When using particles, this information is usually:

- **x**: Rotation angle in radians.
- **y**: Phase during lifetime (0 to 1).
- **z**: Animation frame.

This allows you to easily adjust the shader to a particle system using default particles material. When writing a custom particle shader, this value can be used as desired.

Built-in	Description
in vec2 **VIEWPORT_SIZE**	Size of viewport (in pixels).
in mat4 **VIEW_MATRIX**	World space to view space transform.

Built-in	Description
in mat4 **INV_VIEW_MATRIX**	View space to world space transform.
in mat4 **MAIN_CAM_INV_VIEW_MATRIX**	View space to world space transform of camera used to draw the current viewport.
in mat4 **INV_PROJECTION_MATRIX**	Clip space to view space transform.
in vec3 **NODE_POSITION_WORLD**	Node world space position.
in vec3 **NODE_POSITION_VIEW**	Node view space position.
in vec3 **CAMERA_POSITION_WORLD**	Camera world space position.
in vec3 **CAMERA_DIRECTION_WORLD**	Camera world space direction.
in uint **CAMERA_VISIBLE_LAYERS**	Cull layers of the camera rendering the current pass.

Built-in	Description
in bool **OUTPUT_IS_SRGB**	`true` when output is in sRGB color space (this is `true` in the Compatibility renderer, `false` in Forward+ and Forward Mobile).
in int **INSTANCE_ID**	Instance ID for instancing.
in vec4 **INSTANCE_CUSTOM**	Instance custom data (for particles, mostly).
in int **VIEW_INDEX**	The view that we are rendering. `VIEW_MONO_LEFT` (`0`) for Mono (not multiview) or left eye, `VIEW_RIGHT` (`1`) for right eye.
in int **VIEW_MONO_LEFT**	Constant for Mono or left eye, always `0`.
in int **VIEW_RIGHT**	Constant for right eye, always `1`.
in vec3 **EYE_OFFSET**	Position offset for the eye being rendered. Only applicable for multiview rendering.

Built-in	**Description**
inout vec3 **VERTEX**	Vertex in local coordinates.
in int **VERTEX_ID**	The index of the current vertex in the vertex buffer.
inout vec3 **NORMAL**	Normal in local coordinates.
inout vec3 **TANGENT**	Tangent in local coordinates.
inout vec3 **BINORMAL**	Binormal in local coordinates.
out vec4 **POSITION**	If written to, overrides final vertex position.
inout vec2 **UV**	UV main channel.
inout vec2 **UV2**	UV secondary channel.
inout vec4 **COLOR**	Color from vertices.
out float **ROUGHNESS**	Roughness for vertex lighting.

Built-in	Description
inout float **POINT_SIZE**	Point size for point rendering.
inout mat4 **MODELVIEW_MATRIX**	Model space to view space transform (use if possible).
inout mat3 **MODELVIEW_NORMAL_MATRIX**	
in mat4 **MODEL_MATRIX**	Model space to world space transform.
in mat3 **MODEL_NORMAL_MATRIX**	
inout mat4 **PROJECTION_MATRIX**	View space to clip space transform.
in uvec4 **BONE_INDICES**	
in vec4 **BONE_WEIGHTS**	
in vec4 **CUSTOM0**	Custom value from vertex primitive. When using extra

Built-in	Description
	UVs, `xy` is UV3 and `zw` is UV4.
in vec4 **CUSTOM1**	Custom value from vertex primitive. When using extra UVs, `xy` is UV5 and `zw` is UV6.
in vec4 **CUSTOM2**	Custom value from vertex primitive. When using extra UVs, `xy` is UV7 and `zw` is UV8.
in vec4 **CUSTOM3**	Custom value from vertex primitive.

Note

`MODELVIEW_MATRIX` combines both the `MODEL_MATRIX` and `VIEW_MATRIX` and is better suited when floating point issues may arise. For example, if the object is very far away from the world origin, you may run into floating point issues when using the separated `MODEL_MATRIX` and `VIEW_MATRIX`.

Note

`INV_VIEW_MATRIX` is the matrix used for rendering the object in that pass, not like `MAIN_CAM_INV_VIEW_MATRIX`, which is the matrix of the camera in the scene. In the shadow pass, `INV_VIEW_MATRIX`'s view is based on the camera that is located at the position of the light.

Fragment built-ins

The default use of a Godot fragment processor function is to set up the material properties of your object and to let the built-in renderer handle the final shading. However, you are not required to use all these properties, and if you don't write to them, Godot will optimize away the corresponding functionality.

Built-in	Description
in vec2 **VIEWPORT_SIZE**	Size of viewport (in pixels).
in vec4 **FRAGCOORD**	Coordinate of pixel center in screen space. `xy` specifies position in window, `z` specifies fragment depth if `DEPTH` is not used. Origin is lower-left.
in bool **FRONT_FACING**	`true` if current face is front facing.
in vec3 **VIEW**	Normalized vector from fragment position to camera (in view space). This is the

Built-in	Description
	same for both perspective and orthogonal cameras.
in vec2 **UV**	UV that comes from vertex function.
in vec2 **UV2**	UV2 that comes from vertex function.
in vec4 **COLOR**	COLOR that comes from vertex function.
in vec2 **POINT_COORD**	Point Coordinate for drawing points with POINT_SIZE.
in bool **OUTPUT_IS_SRGB**	`true` when output is in sRGB color space (this is `true` in the Compatibility renderer, `false` in Forward+ and Forward Mobile).
in mat4 **MODEL_MATRIX**	Model space to world space transform.
in mat3 **MODEL_NORMAL_MATRIX**	

Built-in	Description
in mat4 **VIEW_MATRIX**	World space to view space transform.
in mat4 **INV_VIEW_MATRIX**	View space to world space transform.
in mat4 **PROJECTION_MATRIX**	View space to clip space transform.
in mat4 **INV_PROJECTION_MATRIX**	Clip space to view space transform.
in vec3 **NODE_POSITION_WORLD**	Node position, in world space.
in vec3 **NODE_POSITION_VIEW**	Node position, in view space.
in vec3 **CAMERA_POSITION_WORLD**	Camera position, in world space.
in vec3 **CAMERA_DIRECTION_WORLD**	Camera direction, in world space.
in uint **CAMERA_VISIBLE_LAYERS**	Cull layers of the camera rendering the current pass.

Built-in	**Description**
in vec3 **VERTEX**	Vertex that comes from vertex function (default, in view space).
inout vec3 **LIGHT_VERTEX**	A writable version of `VERTEX` that can be used to alter light and shadows. Writing to this will not change the position of the fragment.
in int **VIEW_INDEX**	The view that we are rendering. `VIEW_MONO_LEFT` (`0`) for Mono (not multiview) or left eye, `VIEW_RIGHT` (`1`) for right eye.
in int **VIEW_MONO_LEFT**	Constant for Mono or left eye, always `0`.
in int **VIEW_RIGHT**	Constant for right eye, always `1`.
in vec3 **EYE_OFFSET**	Position offset for the eye being rendered. Only applicable for multiview rendering.

Built-in	Description
sampler2D **SCREEN_TEXTURE**	Removed in Godot 4. Use a `sampler2D` with `hint_screen_texture` instead.
in vec2 **SCREEN_UV**	Screen UV coordinate for current pixel.
sampler2D **DEPTH_TEXTURE**	Removed in Godot 4. Use a `sampler2D` with `hint_depth_texture` instead.
out float **DEPTH**	Custom depth value (0..1). If `DEPTH` is being written to in any shader branch, then you are responsible for setting the `DEPTH` for **all** other branches. Otherwise, the graphics API will leave them uninitialized.
inout vec3 **NORMAL**	Normal that comes from vertex function (default, in view space).
inout vec3 **TANGENT**	Tangent that comes from vertex function.

Built-in	Description
inout vec3 **BINORMAL**	Binormal that comes from vertex function.
out vec3 **NORMAL_MAP**	Set normal here if reading normal from a texture instead of NORMAL.
out float **NORMAL_MAP_DEPTH**	Depth from variable above. Defaults to 1.0.
out vec3 **ALBEDO**	Albedo (default white).
out float **ALPHA**	Alpha (0..1); if read from or written to, the material will go to the transparent pipeline.
out float **ALPHA_SCISSOR_THRESHOLD**	If written to, values below a certain amount of alpha are discarded.
out float **ALPHA_HASH_SCALE**	
out float **ALPHA_ANTIALIASING_EDGE**	

Built-in	Description
out vec2 **ALPHA_TEXTURE_COORDINATE**	
out float **METALLIC**	Metallic (0..1).
out float **SPECULAR**	Specular. Defaults to 0.5, best not to modify unless you want to change IOR.
out float **ROUGHNESS**	Roughness (0..1).
out float **RIM**	Rim (0..1). If used, Godot calculates rim lighting.
out float **RIM_TINT**	Rim Tint, goes from 0 (white) to 1 (albedo). If used, Godot calculates rim lighting.
out float **CLEARCOAT**	Small added specular blob. If used, Godot calculates Clearcoat.
out float **CLEARCOAT_GLOSS**	Gloss of Clearcoat. If used, Godot calculates Clearcoat.

Built-in	Description
out float **ANISOTROPY**	For distorting the specular blob according to tangent space.
out vec2 **ANISOTROPY_FLOW**	Distortion direction, use with flowmaps.
out float **SSS_STRENGTH**	Strength of Subsurface Scattering. If used, Subsurface Scattering will be applied to object.
out vec4 **SSS_TRANSMITTANCE_COLOR**	
out float **SSS_TRANSMITTANCE_DEPTH**	
out float **SSS_TRANSMITTANCE_BOOST**	
inout vec3 **BACKLIGHT**	
out float **AO**	Strength of Ambient Occlusion. For use with pre-baked AO.

Built-in	Description
out float **AO_LIGHT_AFFECT**	How much AO affects lights (0..1; default 0).
out vec3 **EMISSION**	Emission color (can go over 1,1,1 for HDR).
out vec4 **FOG**	If written to, blends final pixel color with FOG.rgb based on FOG.a.
out vec4 **RADIANCE**	If written to, blends environment map radiance with RADIANCE.rgb based on RADIANCE.a.
out vec4 **IRRADIANCE**	If written to, blends environment map IRRADIANCE with IRRADIANCE.rgb based on IRRADIANCE.a.

Note

Shaders going through the transparent pipeline when `ALPHA` is written to may exhibit transparency sorting issues. Read the transparency sorting section in the 3D rendering limitations page for more information and ways to avoid issues.

Light built-ins

Writing light processor functions is completely optional. You can skip the light function by setting render_mode to `unshaded`. If no light function is written, Godot will use the material properties written to in the fragment function to calculate the lighting for you (subject to the render_mode).

The light function is called for every light in every pixel. It is called within a loop for each light type.

Below is an example of a custom light function using a Lambertian lighting model:

```
void light() {
    DIFFUSE_LIGHT += clamp(dot(NORMAL, LIGHT), 0.0, 1.0) *
ATTENUATION * LIGHT_COLOR;
}
```

If you want the lights to add together, add the light contribution to `DIFFUSE_LIGHT` using `+=`, rather than overwriting it.

Warning

The `light()` function won't be run if the `vertex_lighting` render mode is enabled, or if **Rendering > Quality > Shading > Force Vertex Shading** is enabled in the Project Settings. (It's enabled by default on mobile platforms.)

Built-in	Description
in vec2 **VIEWPORT_SIZE**	Size of viewport (in pixels).

Built-in	Description
in vec4 **FRAGCOORD**	Coordinate of pixel center in screen space. `xy` specifies position in window, `z` specifies fragment depth if `DEPTH` is not used. Origin is lower-left.
in mat4 **MODEL_MATRIX**	Model space to world space transform.
in mat4 **INV_VIEW_MATRIX**	View space to world space transform.
in mat4 **VIEW_MATRIX**	World space to view space transform.
in mat4 **PROJECTION_MATRIX**	View space to clip space transform.
in mat4 **INV_PROJECTION_MATRIX**	Clip space to view space transform.
in vec3 **NORMAL**	Normal vector, in view space.
in vec2 **UV**	UV that comes from vertex function.

Built-in	Description
in vec2 **UV2**	UV2 that comes from vertex function.
in vec3 **VIEW**	View vector, in view space.
in vec3 **LIGHT**	Light Vector, in view space.
in vec3 **LIGHT_COLOR**	Color of light multiplied by `energy * PI`. The `PI` multiplication is present because physically-based lighting models include a division by `PI`.
in float **SPECULAR_AMOUNT**	2.0 * `light_specular` property for `OmniLight3D` and `SpotLight3D`. 1.0 for `DirectionalLight3D`.
in bool **LIGHT_IS_DIRECTIONAL**	`true` if this pass is a `DirectionalLight3D`.
in float **ATTENUATION**	Attenuation based on distance or shadow.
in vec3 **ALBEDO**	Base albedo.

Built-in	Description
in vec3 **BACKLIGHT**	
in float **METALLIC**	Metallic.
in float **ROUGHNESS**	Roughness.
in bool **OUTPUT_IS_SRGB**	`true` when output is in sRGB color space (this is `true` in the Compatibility renderer, `false` in Forward+ and Forward Mobile).
out vec3 **DIFFUSE_LIGHT**	Diffuse light result.
out vec3 **SPECULAR_LIGHT**	Specular light result.
out float **ALPHA**	Alpha (0..1); if written to, the material will go to the transparent pipeline.

Note

Shaders going through the transparent pipeline when `ALPHA` is written to may exhibit transparency sorting issues. Read the transparency sorting section in the 3D rendering limitations page for more information and ways to avoid issues.

Transparent materials also cannot cast shadows or appear in `hint_screen_texture` and `hint_depth_texture` uniforms. This in turn prevents those materials from appearing in screen-space reflections or refraction. SDFGI sharp reflections are not visible on transparent materials (only rough reflections are visible on transparent materials).

CanvasItem shaders

CanvasItem shaders are used to draw all 2D elements in Godot. These include all nodes that inherit from CanvasItems, and all GUI elements.

CanvasItem shaders contain less built-in variables and functionality than Spatial shaders, but they maintain the same basic structure with vertex, fragment, and light processor functions.

Render modes

Render mode	Description
blend_mix	Mix blend mode (alpha is transparency), default.
blend_add	Additive blend mode.
blend_sub	Subtractive blend mode.
blend_mul	Multiplicative blend mode.
blend_premul_alpha	Pre-multiplied alpha blend mode.
blend_disabled	Disable blending, values (including alpha) are written as-is.

Render mode	Description
unshaded	Result is just albedo. No lighting/shading happens in material.
light_only	Only draw on light pass.
skip_vertex_transform	VERTEX needs to be transformed manually in vertex function.
world_vertex_coords	VERTEX is modified in world coordinates instead of local.

Built-ins

Values marked as "in" are read-only. Values marked as "out" are for optional writing and will not necessarily contain sensible values. Values marked as "inout" provide a sensible default value, and can optionally be written to. Samplers are not subjects of writing and they are not marked.

Global built-ins

Global built-ins are available everywhere, including custom functions.

Built-in	Description
in float **TIME**	Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's not affected by time_scale or pausing. If you need a `TIME` variable that can be scaled or paused, add your own global shader uniform and update it each frame.
in float **PI**	A `PI` constant (`3.141592`). A ration of circle's circumference to its diameter and amount of radians in half turn.
in float **TAU**	A `TAU` constant (`6.283185`). An equivalent of `PI * 2` and amount of radians in full turn.
in float **E**	An `E` constant (`2.718281`). Euler's number and a base of the natural logarithm.

Vertex built-ins

Vertex data (`VERTEX`) is presented in local space (pixel coordinates, relative to the Node2D's origin). If not written to, these values will not be modified and be passed through as they came.

The user can disable the built-in model to world transform (world to screen and projection will still happen later) and do it manually with the following code:

```
shader_type canvas_item;
render_mode skip_vertex_transform;

void vertex() {

    VERTEX = (MODEL_MATRIX * vec4(VERTEX, 0.0, 1.0)).xy;
}
```

Other built-ins, such as UV and COLOR, are also passed through to the fragment function if not modified.

For instancing, the INSTANCE_CUSTOM variable contains the instance custom data. When using particles, this information is usually:

- **x**: Rotation angle in radians.
- **y**: Phase during lifetime (0 to 1).
- **z**: Animation frame.

Built-in	Description
in mat4 **MODEL_MATRIX**	Local space to world space transform. World space is the coordinates you normally use in the editor.
in mat4 **CANVAS_MATRIX**	World space to canvas space transform. In canvas space the origin is the upper-left corner of the screen and coordinates ranging from (0, 0) to viewport size.
in mat4 **SCREEN_MATRIX**	Canvas space to clip space. In clip space coordinates ranging from (-1, -1) to (1, 1).

Built-in	Description
in int **INSTANCE_ID**	Instance ID for instancing.
in vec4 **INSTANCE_CUSTOM**	Instance custom data.
in bool **AT_LIGHT_PASS**	Always `false`.
in vec2 **TEXTURE_PIXEL_SIZE**	Normalized pixel size of default 2D texture. For a Sprite2D with a texture of size 64x32px, **TEXTURE_PIXEL_SIZE** = `vec2(1/64, 1/32)`
inout vec2 **VERTEX**	Vertex, in local space.
in int **VERTEX_ID**	The index of the current vertex in the vertex buffer.
inout vec2 **UV**	Normalized texture coordinates. Range from 0 to 1.
inout vec4 **COLOR**	Color from vertex primitive.
inout float **POINT_SIZE**	Point size for point drawing.

Built-in	Description
in vec4 **CUSTOM0**	Custom value from vertex primitive.
in vec4 **CUSTOM1**	Custom value from vertex primitive.

Fragment built-ins

Certain Nodes (for example, Sprite2Ds) display a texture by default. However, when a custom fragment function is attached to these nodes, the texture lookup needs to be done manually. Godot provides the texture color in the `COLOR` built-in variable multiplied by the node's color. To read the texture color by itself, you can use:

```
COLOR = texture(TEXTURE, UV);
```

Similarly, if a normal map is used in the CanvasTexture, Godot uses it by default and assigns its value to the built-in `NORMAL` variable. If you are using a normal map meant for use in 3D, it will appear inverted. In order to use it in your shader, you must assign it to the `NORMALMAP` property. Godot will handle converting it for use in 2D and overwriting `NORMAL`.

```
NORMALMAP = texture(NORMAL_TEXTURE, UV).rgb;
```

Built-in	Description
in vec4 **FRAGCOORD**	Coordinate of pixel center. In screen space. `xy` specifies position in

Built-in	Description
	window. Origin is upper-left.
in vec2 **SCREEN_PIXEL_SIZE**	Size of individual pixels. Equal to inverse of resolution.
in vec2 **POINT_COORD**	Coordinate for drawing points.
sampler2D **TEXTURE**	Default 2D texture.
in vec2 **TEXTURE_PIXEL_SIZE**	Normalized pixel size of default 2D texture. For a Sprite2D with a texture of size 64x32px, **TEXTURE_PIXEL_SIZE** = `vec2(1/64, 1/32)`
in bool **AT_LIGHT_PASS**	Always `false`.
sampler2D **SPECULAR_SHININESS_TEXTURE**	Specular shininess texture of this object.
in vec4 **SPECULAR_SHININESS**	Specular shininess color, as sampled from the texture.

Built-in	Description
in vec2 **UV**	UV from vertex function.
in vec2 **SCREEN_UV**	Screen UV coordinate for current pixel.
sampler2D **SCREEN_TEXTURE**	Removed in Godot 4. Use a `sampler2D` with `hint_screen_texture` instead.
inout vec3 **NORMAL**	Normal read from **NORMAL_TEXTURE**. Writable.
sampler2D **NORMAL_TEXTURE**	Default 2D normal texture.
out vec3 **NORMAL_MAP**	Configures normal maps meant for 3D for use in 2D. If used, overrides **NORMAL**.
out float **NORMAL_MAP_DEPTH**	Normalmap depth for scaling.
inout vec2 **VERTEX**	Pixel position in screen space.

Built-in	Description
inout vec2 **SHADOW_VERTEX**	Same as `VERTEX` but can be written to alter shadows.
inout vec3 **LIGHT_VERTEX**	Same as `VERTEX` but can be written to alter lighting. Z component represents height.
inout vec4 **COLOR**	Color from vertex function multiplied by the **TEXTURE** color. Also output color value.

Light built-ins

Light processor functions work differently in Godot 4.x than they did in Godot 3.x. In Godot 4.x all lighting is done during the regular draw pass. In other words, Godot no longer draws the object again for each light.

Use render_mode `unshaded` if you do not want the light processor function to run. Use render_mode `light_only` if you only want to see the impact of lighting on an object; this can be useful when you only want the object visible where it is covered by light.

If you define a light function it will replace the built in light function, even if your light function is empty.

Below is an example of a light shader that takes a CanvasItem's normal map into account:

```
void light() {
  float cNdotL = max(0.0, dot(NORMAL, LIGHT_DIRECTION));
  LIGHT = vec4(LIGHT_COLOR.rgb * COLOR.rgb * LIGHT_ENERGY *
cNdotL, LIGHT_COLOR.a);
}
```

Built-in	Description
in vec4 **FRAGCOORD**	Coordinate of pixel center. In screen space. `xy` specifies position in window. Origin is lower-left.
in vec3 **NORMAL**	Input Normal.
in vec4 **COLOR**	Input Color. This is the output of the fragment function.
in vec2 **UV**	UV from vertex function, equivalent to the UV in the fragment function.
sampler2D **TEXTURE**	Current texture in use for CanvasItem.
in vec2 **TEXTURE_PIXEL_SIZE**	Normalized pixel size of **TEXTURE**. For a Sprite2D with a **TEXTURE** of size 64x32px, **TEXTURE_PIXEL_SIZE** = `vec2(1/64, 1/32)`

Built-in	Description
in vec2 **SCREEN_UV**	Screen UV coordinate for current pixel.
in vec2 **POINT_COORD**	UV for Point Sprite.
in vec4 **LIGHT_COLOR**	Color of Light multiplied by Light's texture.
in float **LIGHT_ENERGY**	Energy multiplier of Light.
in vec3 **LIGHT_POSITION**	Position of Light in screen space. If using a `DirectionalLight2D` this is always `vec3(0,0,0)`.
in vec3 **LIGHT_DIRECTION**	Direction of Light in screen space.
in bool **LIGHT_IS_DIRECTIONAL**	`true` if this pass is a `DirectionalLight2D`.
in vec3 **LIGHT_VERTEX**	Pixel position, in screen space as modified in the fragment function.
inout vec4 **LIGHT**	Output color for this Light.

Built-in	Description
in vec4 **SPECULAR_SHININESS**	Specular shininess, as set in the object's texture.
out vec4 **SHADOW_MODULATE**	Multiply shadows cast at this point by this color.

SDF functions

There are a few additional functions implemented to sample an automatically generated Signed Distance Field texture. These functions available for Fragment and Light functions of CanvasItem shaders.

The signed distance field is generated from LightOccluder2D nodes present in the scene with the **SDF Collision** property enabled (which is the default). See the 2D lights and shadows documentation for more information.

Function	Description
float **texture_sdf** (vec2 sdf_pos)	Performs an SDF texture lookup.
vec2 **texture_sdf_normal** (vec2 sdf_pos)	Calculates a normal from the SDF texture.

Function	Description
vec2 **sdf_to_screen_uv** (vec2 sdf_pos)	Converts an SDF to screen UV.
vec2 **screen_uv_to_sdf** (vec2 uv)	Converts screen UV to an SDF.

Particle shaders

Particle shaders are a special type of shader that runs before the object is drawn. They are used for calculating material properties such as color, position, and rotation. They are drawn with any regular material for CanvasItem or Spatial, depending on whether they are 2D or 3D.

Particle shaders are unique because they are not used to draw the object itself; they are used to calculate particle properties, which are then used by a CanvasItem or Spatial shader. They contain two processor functions: `start()` and `process()`.

Unlike other shader types, particle shaders keep the data that was output the previous frame. Therefore, particle shaders can be used for complex effects that take place over multiple frames.

Note

Particle shaders are only available with GPU-based particle nodes (GPUParticles2D and GPUParticles3D).

CPU-based particle nodes (CPUParticles2D and CPUParticles3D) are *rendered* on the GPU (which means they can use custom CanvasItem or Spatial shaders), but their motion is *simulated* on the CPU.

Render modes

Render mode	Description
keep_data	Do not clear previous data on restart.
disable_force	Disable attractor force.
disable_velocity	Ignore **VELOCITY** value.
collision_use_scale	Scale the particle's size for collisions.

Built-ins

Values marked as "in" are read-only. Values marked as "out" are for optional writing and will not necessarily contain sensible values. Values marked as "inout" provide a sensible default value, and can optionally be written to. Samplers are not subjects of writing and they are not marked.

Global built-ins

Global built-ins are available everywhere, including custom functions.

Built-in	Description
in float **TIME**	Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's not affected by

Built-in	Description
	time_scale or pausing. If you need a `TIME` variable that can be scaled or paused, add your own global shader uniform and update it each frame.
in float **PI**	A `PI` constant (`3.141592`). A ration of circle's circumference to its diameter and amount of radians in half turn.
in float **TAU**	A `TAU` constant (`6.283185`). An equivalent of `PI * 2` and amount of radians in full turn.
in float **E**	An `E` constant (`2.718281`). Euler's number and a base of the natural logarithm.

Start and Process built-ins

These properties can be accessed from both the `start()` and `process()` functions.

Function	Description
in float **LIFETIME**	Particle lifetime.
in float **DELTA**	Delta process time.

Function	**Description**
in uint **NUMBER**	Unique number since emission start.
in uint **INDEX**	Particle index (from total particles).
in mat4 **EMISSION_TRANSFORM**	Emitter transform (used for non-local systems).
in uint **RANDOM_SEED**	Random seed used as base for random.
inout bool **ACTIVE**	`true` when the particle is active, can be set `false`.
inout vec4 **COLOR**	Particle color, can be written to and accessed in mesh's vertex function.
inout vec3 **VELOCITY**	Particle velocity, can be modified.
inout mat4 **TRANSFORM**	Particle transform.
inout vec4 **CUSTOM**	Custom particle data. Accessible from shader of mesh as **INSTANCE_CUSTOM**.

Function	Description
inout float **MASS**	Particle mass, intended to be used with attractors. Equals `1.0` by default.
in vec4 **USERDATAX**	Vector that enables the integration of supplementary user-defined data into the particle process shader. `USERDATAX` are six built-ins identified by number, `X` can be numbers between 1 and 6.
in uint **FLAG_EMIT_POSITION**	A flag for using on the last argument of `emit_subparticle` function to assign a position to a new particle's transform.
in uint **FLAG_EMIT_ROT_SCALE**	A flag for using on the last argument of `emit_subparticle` function to assign the rotation and scale to a new particle's transform.
in uint **FLAG_EMIT_VELOCITY**	A flag for using on the last argument of `emit_subparticle` function to assign a velocity to a new particle.
in uint **FLAG_EMIT_COLOR**	A flag for using on the last argument of `emit_subparticle` function to assign a color to a new particle.

Function	Description
in uint **FLAG_EMIT_CUSTOM**	A flag for using on the last argument of `emit_subparticle` function to assign a custom data vector to a new particle.
in vec3 **EMITTER_VELOCITY**	Velocity of the Particles node.
in float **INTERPOLATE_TO_END**	Value of `interp_to_end` property of Particles node.
in uint **AMOUNT_RATIO**	Value of `amount_ratio` property of Particles node.

Note

In order to use the `COLOR` variable in a StandardMaterial3D, set `vertex_color_use_as_albedo` to `true`. In a ShaderMaterial, access it with the `COLOR` variable.

Start built-ins

Built-in	**Description**
in bool **RESTART_POSITION**	`true` if particle is restarted, or emitted without a custom position (i.e. this particle was created by `emit_subparticle()` without the `FLAG_EMIT_POSITION` flag).
in bool **RESTART_ROT_SCALE**	`true` if particle is restarted, or emitted without a custom rotation or scale (i.e. this particle was created by `emit_subparticle()` without the `FLAG_EMIT_ROT_SCALE` flag).
in bool **RESTART_VELOCITY**	`true` if particle is restarted, or emitted without a custom velocity (i.e. this particle was created by `emit_subparticle()` without the `FLAG_EMIT_VELOCITY` flag).
in bool **RESTART_COLOR**	`true` if particle is restarted, or emitted without a custom color (i.e. this particle was created by `emit_subparticle()` without the `FLAG_EMIT_COLOR` flag).
in bool **RESTART_CUSTOM**	`true` if particle is restarted, or emitted without a custom property (i.e. this particle was created by `emit_subparticle()` without the `FLAG_EMIT_CUSTOM` flag).

Built-in	Description

Process built-ins

Built-in	Description
in bool **RESTART**	`true` if the current process frame is first for the particle.
in bool **COLLIDED**	`true` when the particle has collided with a particle collider.
in vec3 **COLLISION_NORMAL**	A normal of the last collision. If there is no collision detected it is equal to `vec3(0.0)`.
in float **COLLISION_DEPTH**	A length of normal of the last collision. If there is no collision detected it is equal to `0.0`.
in vec3 **ATTRACTOR_FORCE**	A combined force of the attractors at the moment on that particle.

Process functions

`emit_subparticle` is currently the only custom function supported by particles shaders. It allows users to add a new particle with specified parameters from a sub-emitter. The newly created particle will only use the properties that match the `flags` parameter. For example, the following code will emit a particle with a specified position, velocity, and color, but unspecified rotation, scale, and custom value:

```
mat4 custom_transform = mat4(1.0);
custom_transform[3].xyz = vec3(10.5, 0.0, 4.0);
emit_subparticle(custom_transform, vec3(1.0, 0.5, 1.0),
vec4(1.0, 0.0, 0.0, 1.0), vec4(1.0), FLAG_EMIT_POSITION |
FLAG_EMIT_VELOCITY | FLAG_EMIT_COLOR);
```

Function	Description
bool **emit_subparticle** (mat4 xform, vec3 velocity, vec4 color, vec4 custom, uint flags)	Emits a particle from a sub-emitter.

Sky shaders

Sky shaders are a special type of shader used for drawing sky backgrounds and for updating radiance cubemaps which are used for image-based lighting (IBL). Sky shaders only have one processing function, the `sky()` function.

There are three places the sky shader is used.

- First the sky shader is used to draw the sky when you have selected to use a Sky as the background in your scene.
- Second, the sky shader is used to update the radiance cubemap when using the Sky for ambient color or reflections.
- Third, the sky shader is used to draw the lower res subpasses which can be used in the high-res background or cubemap pass.

In total, this means the sky shader can run up to six times per frame, however, in practice it will be much less than that because the radiance cubemap does not need to be updated every frame, and not all subpasses will be used. You can change the behavior of the shader based on where it is called by checking the `AT_*_PASS` booleans. For example:

```
shader_type sky;

void sky() {
    if (AT_CUBEMAP_PASS) {
        // Sets the radiance cubemap to a nice shade of blue instead of doing
        // expensive sky calculations
        COLOR = vec3(0.2, 0.6, 1.0);
    } else {
        // Do expensive sky calculations for background sky only
        COLOR = get_sky_color(EYEDIR);
    }
}
```

When using the sky shader to draw a background, the shader will be called for all non-occluded fragments on the screen. However, for the background's subpasses, the shader will be called for every pixel of the subpass.

When using the sky shader to update the radiance cubemap, the sky shader will be called for every pixel in the cubemap. On the other hand, the shader will only be called when the radiance cubemap needs to be updated. The radiance cubemap needs to be updated when any of the shader parameters are updated. For example, if `TIME` is used in the shader, then the radiance cubemap will update every frame. The following list of changes force an update of the radiance cubemap:

- `TIME` is used.
- `POSITION` is used and the camera position changes.
- If any `LIGHTX_*` properties are used and any DirectionalLight3D changes.
- If any uniform is changed in the shader.
- If the screen is resized and either of the subpasses are used.

Try to avoid updating the radiance cubemap needlessly. If you do need to update the radiance cubemap each frame, make sure your Sky process mode is set to REALTIME.

Note that the process mode only affects the rendering of the radiance cubemap. The visible sky is always rendered by calling the fragment shader for every pixel. With complex fragment shaders, this can result in a high rendering overhead. If the sky is static (the conditions listed above are met) or changes slowly, running the full fragment shader every frame is not needed. This can be avoided by rendering the full sky into the radiance cubemap, and reading from this cubemap when rendering the visible sky. With a completely static sky, this means that it needs to be rendered only once.

The following code renders the full sky into the radiance cubemap and reads from that cubemap for displaying the visible sky:

```
shader_type sky;

void sky() {
    if (AT_CUBEMAP_PASS) {
        vec3 dir = EYEDIR;

        vec4 col = vec4(0.0);

        // Complex color calculation

        COLOR = col.xyz;
        ALPHA = 1.0;
    } else {
        COLOR = texture(RADIANCE, EYEDIR).rgb;
    }
}
```

This way, the complex calculations happen only in the cubemap pass, which can be optimized by setting the sky's process mode and the radiance size to get the desired balance between performance and visual fidelity.

Render modes

Subpasses allow you to do more expensive calculations at a lower resolution to speed up your shaders. For example the following code renders clouds at a lower resolution than the rest of the sky:

```
shader_type sky;
render_mode use_half_res_pass;

void sky() {
    if (AT_HALF_RES_PASS) {
        // Run cloud calculation for 1/4 of the pixels
        vec4 color = generate_clouds(EYEDIR);
        COLOR = color.rgb;
        ALPHA = color.a;
    } else {
        // At full resolution pass, blend sky and clouds together
        vec3 color = generate_sky(EYEDIR);
        COLOR = color + HALF_RES_COLOR.rgb * HALF_RES_COLOR.a;
    }
}
```

Render mode	Description
use_half_res_pass	Allows the shader to write to and access the half resolution pass.
use_quarter_res_pass	Allows the shader to write to and access the quarter resolution pass.
disable_fog	If used, fog will not affect the sky.

Built-ins

Values marked as "in" are read-only. Values marked as "out" are for optional writing and will not necessarily contain sensible values. Samplers cannot be written to so they are not marked.

Global built-ins

Global built-ins are available everywhere, including in custom functions.

There are 4 `LIGHTX` lights, accessed as `LIGHT0`, `LIGHT1`, `LIGHT2`, and `LIGHT3`.

Built-in	Description
in float **TIME**	Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's not affected by time_scale or pausing. If you need a `TIME` variable that can be scaled or paused, add your own global shader uniform and update it each frame.
in vec3 **POSITION**	Camera position in world space
samplerCube **RADIANCE**	Radiance cubemap. Can only be read from during background pass. Check `!AT_CUBEMAP_PASS` before using.
in bool **AT_HALF_RES_PASS**	Currently rendering to half resolution pass.
in bool **AT_QUARTER_RES_PASS**	Currently rendering to quarter resolution pass.
in bool **AT_CUBEMAP_PASS**	Currently rendering to radiance cubemap.

Built-in	Description
in bool **LIGHTX_ENABLED**	`LightX` is visible and in the scene. If `false`, other light properties may be garbage.
in float **LIGHTX_ENERGY**	Energy multiplier for `LIGHTX`.
in vec3 **LIGHTX_DIRECTION**	Direction that `LIGHTX` is facing.
in vec3 **LIGHTX_COLOR**	Color of `LIGHTX`.
in float **LIGHTX_SIZE**	Angular diameter of `LIGHTX` in the sky. Expressed in radians. For reference, the sun from earth is about .0087 radians (0.5 degrees).
in float **PI**	A `PI` constant (`3.141592`). A ratio of a circle's circumference to its diameter and amount of radians in half turn.
in float **TAU**	A `TAU` constant (`6.283185`). An equivalent of `PI * 2` and amount of radians in full turn.
in float **E**	An `E` constant (`2.718281`). Euler's number and a base of the natural

Built-in	Description
	logarithm.

Sky built-ins

Built-in	Description
in vec3 **EYEDIR**	Normalized direction of current pixel. Use this as your basic direction for procedural effects.
in vec2 **SCREEN_UV**	Screen UV coordinate for current pixel. Used to map a texture to the full screen.
in vec2 **SKY_COORDS**	Sphere UV. Used to map a panorama texture to the sky.
in vec4 **HALF_RES_COLOR**	Color value of corresponding pixel from half resolution pass. Uses linear filter.
in vec4 **QUARTER_RES_COLOR**	Color value of corresponding pixel from quarter resolution pass. Uses linear filter.
out vec3 **COLOR**	Output color.

Built-in	Description
out float **ALPHA**	Output alpha value, can only be used in subpasses.
out vec4 **FOG**	

Fog shaders

Fog shaders are used to define how fog is added (or subtracted) from a scene in a given area. Fog shaders are always used together with FogVolumes and volumetric fog. Fog shaders only have one processing function, the `fog()` function.

The resolution of the fog shaders depends on the resolution of the volumetric fog froxel grid. Accordingly, the level of detail that a fog shader can add depends on how close the FogVolume is to the camera.

Fog shaders are a special form of compute shader that is called once for every froxel that is touched by an axis aligned bounding box of the associated FogVolume. This means that froxels that just barely touch a given FogVolume will still be used.

Built-ins

Values marked as "in" are read-only. Values marked as "out" are for optional writing and will not necessarily contain sensible values. Samplers cannot be written to so they are not marked.

Global built-ins

Global built-ins are available everywhere, including in custom functions.

Built-in	Description
in float **TIME**	Global time since the engine has started, in seconds. It repeats after every 3,600 seconds (which can be changed with the rollover setting). It's not affected by time_scale or pausing. If you need a `TIME` variable that can be scaled or paused, add your own global shader uniform and update it each frame.
in float **PI**	A `PI` constant (`3.141592`). A ratio of a circle's circumference to its diameter and amount of radians in half turn.
in float **TAU**	A `TAU` constant (`6.283185`). An equivalent of `PI * 2` and amount of radians in full turn.
in float **E**	An `E` constant (`2.718281`). Euler's number and a base of the natural logarithm.

Fog built-ins

All of the output values of fog volumes overlap one another. This allows FogVolumes to be rendered efficiently as they can all be drawn at once.

Built-in	**Description**
in vec3 **WORLD_POSITION**	Position of current froxel cell in world space.
in vec3 **OBJECT_POSITION**	Position of the center of the current FogVolume in world space.
in vec3 **UVW**	3-dimensional uv, used to map a 3D texture to the current FogVolume.
in vec3 **SIZE**	Size of the current FogVolume when its shape has a size.
in vec3 **SDF**	Signed distance field to the surface of the FogVolume. Negative if inside volume, positive otherwise.
out vec3 **ALBEDO**	Output base color value, interacts with light to produce final color. Only written to fog volume if used.
out float **DENSITY**	Output density value. Can be negative to allow subtracting one volume from another. Density must be used for fog shader to write anything at all.

Built-in	Description
out vec3 **EMISSION**	Output emission color value, added to color during light pass to produce final color. Only written to fog volume if used.

Your first shader

This tutorial series will walk you through writing your first shader. It is intended for people who have very little prior experience with shaders and want to get started with the basics. This tutorial will not cover advanced topics and it is not comprehensive. For a comprehensive and detailed overview of shaders in Godot see the Shading Reference Page.

The "your first shader" tutorials walk you through the process of writing a shader step-by-step.

For a more general introduction into shaders and the OpenGL Shading Language, use The Book of Shaders [https://thebookofshaders.com].

- Your first 2D shader
- Your first 3D shader
- Your second 3D shader

Your first 2D shader

Introduction

Shaders are special programs that execute on the GPU and are used for rendering graphics. All modern rendering is done with shaders. For a more detailed description of what shaders are please see What are shaders.

This tutorial will focus on the practical aspects of writing shader programs by walking you through the process of writing a shader with both vertex and fragment functions. This tutorial targets absolute beginners to shaders.

Note

If you have experience writing shaders and are just looking for an overview of how shaders work in Godot, see the Shading Reference.

Setup

CanvasItem shaders are used to draw all 2D objects in Godot, while Spatial shaders are used to draw all 3D objects.

In order to use a shader it must be attached inside a Material which must be attached to an object. Materials are a type of Resource. To draw multiple objects with the same material, the material must be attached to each object.

All objects derived from a CanvasItem have a material property. This includes all GUI elements, Sprite2Ds, TileMapLayers, MeshInstance2Ds etc. They also have an option to inherit their parent's material. This can be useful if you have a large number of nodes that you want to use the same material.

To begin, create a Sprite2D node. You can use any CanvasItem, so long as it is drawing to the canvas, so for this tutorial we will use a Sprite2D, as it is the easiest CanvasItem to start drawing with.

In the Inspector, click beside "Texture" where it says "[empty]" and select "Load", then select "icon.svg". For new projects, this is the Godot icon. You should now see the icon in the viewport.

Next, look down in the Inspector, under the CanvasItem section, click beside "Material" and select "New ShaderMaterial". This creates a new Material resource. Click on the sphere that appears. Godot currently doesn't know whether you are writing a CanvasItem Shader or a Spatial Shader and it previews the output of spatial shaders. So what you are seeing is the output of the default Spatial Shader.

Click beside "Shader" and select "New Shader". Finally, click on the shader you just created and the shader editor will open. You are now ready to begin writing your first shader.

Your first CanvasItem shader

In Godot, all shaders start with a line specifying what type of shader they are. It uses the following format:

```
shader_type canvas_item;
```

Because we are writing a CanvasItem shader, we specify `canvas_item` in the first line. All our code will go beneath this declaration.

This line tells the engine which built-in variables and functionality to supply you with.

In Godot you can override three functions to control how the shader operates; `vertex`, `fragment`, and `light`. This tutorial will walk you through writing a shader with both vertex and fragment functions. Light functions are significantly more complex than vertex and fragment functions and so will not be covered here.

Your first fragment function

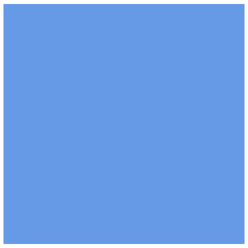
The fragment function runs for every pixel in a Sprite2D and determines what color that pixel should be.

They are restricted to the pixels covered by the Sprite2D, that means you cannot use one to, for example, create an outline around a Sprite2D.

The most basic fragment function does nothing except assign a single color to every pixel.

We do so by writing a `vec4` to the built-in variable `COLOR`. `vec4` is shorthand for constructing a vector with 4 numbers. For more information about vectors see the Vector math tutorial. `COLOR` is both an input variable to the fragment function and the final output from it.

```
void fragment(){
  COLOR = vec4(0.4, 0.6, 0.9, 1.0);
}
```



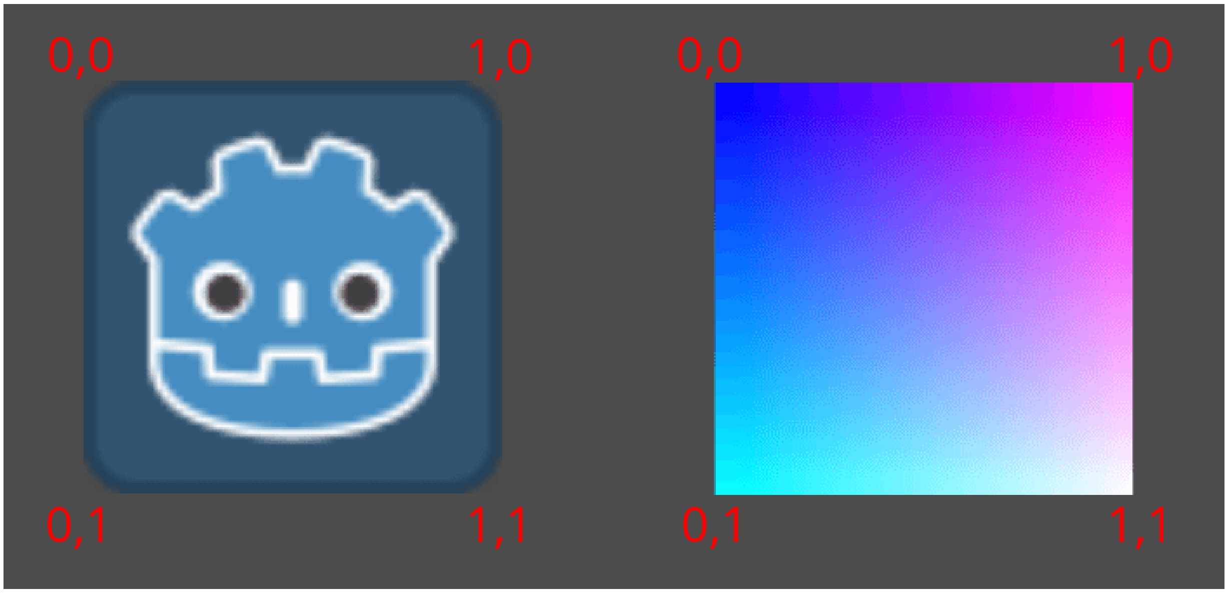
Congratulations! You're done. You have successfully written your first shader in Godot.

Now let's make things more complex.

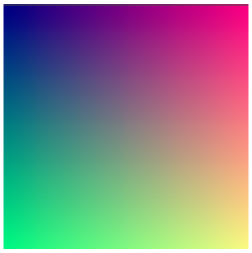
There are many inputs to the fragment function that you can use for calculating `COLOR`. `UV` is one of them. UV coordinates are specified in your Sprite2D (without you knowing it!) and they tell the shader where to read from textures for each part of the mesh.

In the fragment function you can only read from `UV`, but you can use it in other functions or to assign values to `COLOR` directly.

`UV` varies between 0-1 from left-right and from top-bottom.



```
void fragment() {
  COLOR = vec4(UV, 0.5, 1.0);
}
```



Using `TEXTURE` built-in

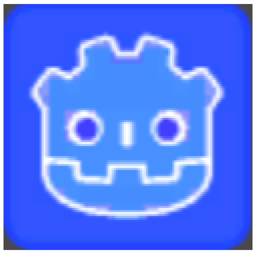
The default fragment function reads from the set Sprite2D texture and displays it.

When you want to adjust a color in a Sprite2D you can adjust the color from the texture manually like in the code below.

```
void fragment(){
  // This shader will result in a blue-tinted icon
  COLOR.b = 1.0;
}
```

Certain nodes, like Sprite2Ds, have a dedicated texture variable that can be accessed in the shader using `TEXTURE`. If you want to use the Sprite2D texture to combine with other colors, you can use the `UV` with the `texture` function to access this variable. Use them to redraw the Sprite2D with the texture.

```
void fragment(){
  COLOR = texture(TEXTURE, UV); // Read from texture again.
  COLOR.b = 1.0; //set blue channel to 1.0
}
```



Uniform input

Uniform input is used to pass data into a shader that will be the same across the entire shader.

You can use uniforms by defining them at the top of your shader like so:

```
uniform float size;
```

For more information about usage see the Shading Language doc.

Add a uniform to change the amount of blue in our Sprite2D.

```
uniform float blue = 1.0; // you can assign a default value to
uniforms

void fragment(){
  COLOR = texture(TEXTURE, UV); // Read from texture
  COLOR.b = blue;
}
```

Now you can change the amount of blue in the Sprite2D from the editor. Look back at the Inspector under where you created your shader. You should see a section called "Shader Param". Unfold that section and you will see the uniform you just declared. If you change the value in the editor, it will overwrite the default value you provided in the shader.

Interacting with shaders from code

You can change uniforms from code using the function `set_shader_parameter()` which is called on the node's material resource. With a Sprite2D node, the following code can be used to set the `blue` uniform.

GDScript

```
var blue_value = 1.0
material.set_shader_parameter("blue", blue_value)
```

C#

```
var blueValue = 1.0;
((ShaderMaterial)Material).SetShaderParameter("blue",
blueValue);
```

Note that the name of the uniform is a string. The string must match exactly with how it is written in the shader, including spelling and case.

Your first vertex function

Now that we have a fragment function, let's write a vertex function.

Use the vertex function to calculate where on the screen each vertex should end up.

The most important variable in the vertex function is `VERTEX`. Initially, it specifies the vertex coordinates in your model, but you also write to it to determine where to actually draw those vertices. `VERTEX` is a `vec2` that is initially presented in local-space (i.e. not relative to the camera, viewport, or parent nodes).

You can offset the vertices by directly adding to `VERTEX`.

```
void vertex() {
  VERTEX += vec2(10.0, 0.0);
}
```

Combined with the `TIME` built-in variable, this can be used for basic animation.

```
void vertex() {
  // Animate Sprite2D moving in big circle around its location
  VERTEX += vec2(cos(TIME)*100.0, sin(TIME)*100.0);
}
```

Conclusion

At their core, shaders do what you have seen so far, they compute `VERTEX` and `COLOR`. It is up to you to dream up more complex mathematical strategies for assigning values to those variables.

For inspiration, take a look at some of the more advanced shader tutorials, and look at other sites like Shadertoy [https://www.shadertoy.com/results?query=&sort=popular&from=10&num=4] and The Book of Shaders [https://thebookofshaders.com].

Your first 3D shader

You have decided to start writing your own custom Spatial shader. Maybe you saw a cool trick online that was done with shaders, or you have found that the StandardMaterial3D isn't quite meeting your needs. Either way, you have decided to write your own and now you need to figure out where to start.

This tutorial will explain how to write a Spatial shader and will cover more topics than the CanvasItem tutorial.

Spatial shaders have more built-in functionality than CanvasItem shaders. The expectation with spatial shaders is that Godot has already provided the functionality for common use cases and all the user needs to do in the shader is set the proper parameters. This is especially true for a PBR (physically based rendering) workflow.

This is a two-part tutorial. In this first part we will create terrain using vertex displacement from a heightmap in the vertex function. In the second part we will take the concepts from this tutorial and set up custom materials in a fragment shader by writing an ocean water shader.

Note

This tutorial assumes some basic shader knowledge such as types (`vec2`, `float`, `sampler2D`), and functions. If you are uncomfortable with these concepts it is best to get a gentle introduction from The Book of Shaders [https://thebookofshaders.com] before completing this tutorial.

Where to assign my material

In 3D, objects are drawn using Meshes. Meshes are a resource type that store geometry (the shape of your object) and materials (the color and how the object reacts to light) in units called "surfaces". A Mesh can have multiple surfaces, or just one. Typically, you would import a mesh from another program (e.g. Blender). But Godot also has a few PrimitiveMeshes that allow you to add basic geometry to a scene without importing Meshes.

There are multiple node types that you can use to draw a mesh. The main one is MeshInstance3D, but you can also use GPUParticles3D, MultiMeshes (with a MultiMeshInstance3D), or others.

Typically, a material is associated with a given surface in a mesh, but some nodes, like MeshInstance3D, allow you to override the material for a specific surface, or for all surfaces.

If you set a material on the surface or mesh itself, then all MeshInstance3Ds that share that mesh will share that material. However, if you want to reuse the same mesh across multiple mesh instances, but have different materials for each instance then you should set the material on the MeshInstance3D.

For this tutorial we will set our material on the mesh itself rather than taking advantage of the MeshInstance3D's ability to override materials.

Setting up

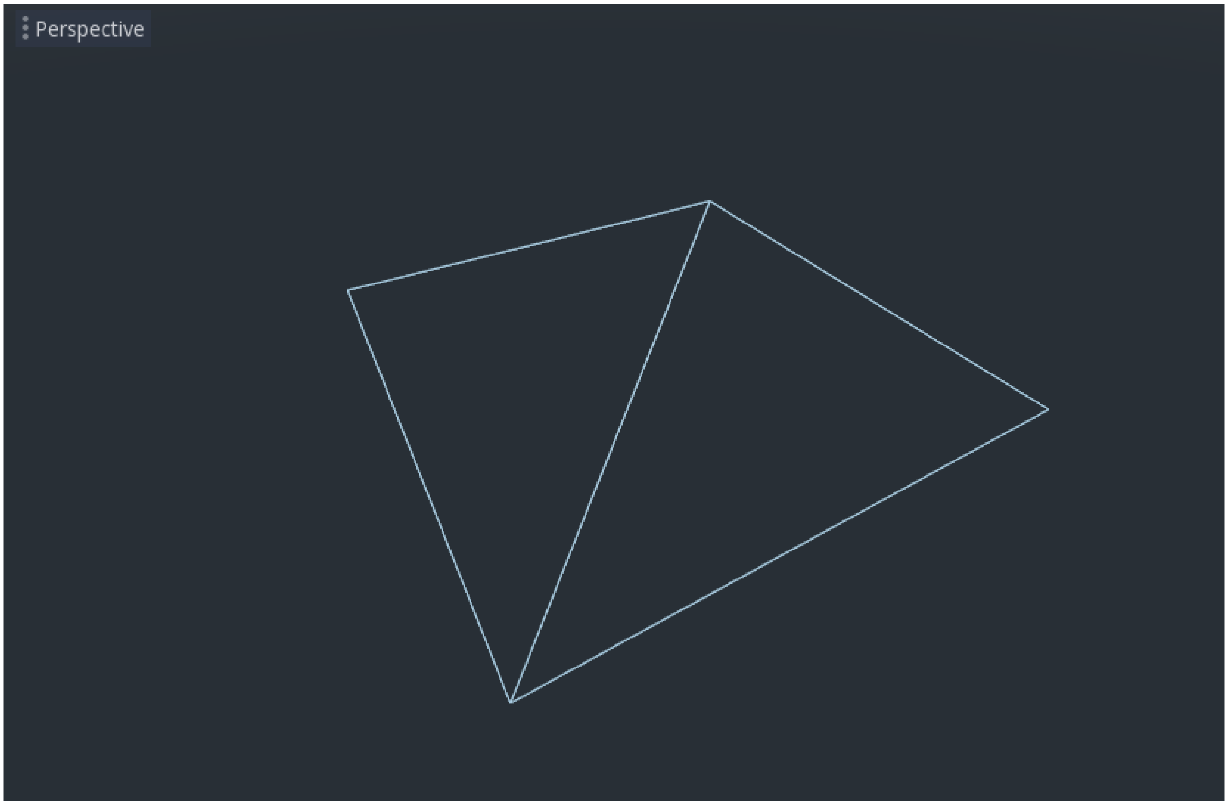
Add a new MeshInstance3D node to your scene.

In the inspector tab beside "Mesh" click "[empty]" and select "New PlaneMesh". Then click on the image of a plane that appears.

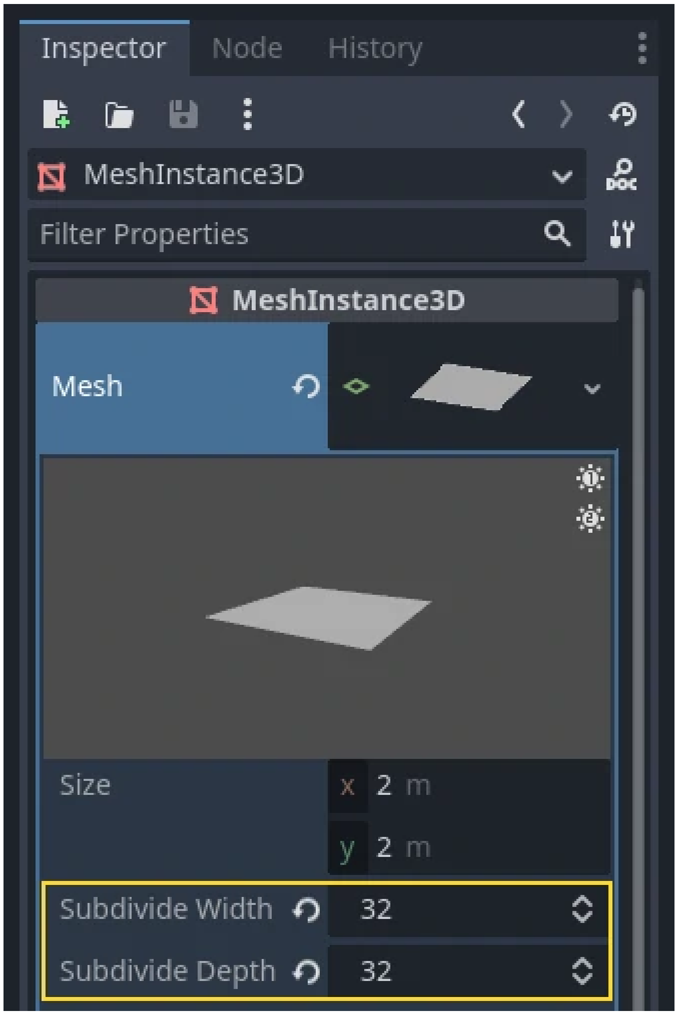
This adds a PlaneMesh to our scene.

Then, in the viewport, click in the upper left corner on the button that says "Perspective". A menu will appear. In the middle of the menu are options for how to display the scene. Select 'Display Wireframe'.

This will allow you to see the triangles making up the plane.



Now set `Subdivide Width` and `Subdivide Depth` of the PlaneMesh to `32`.

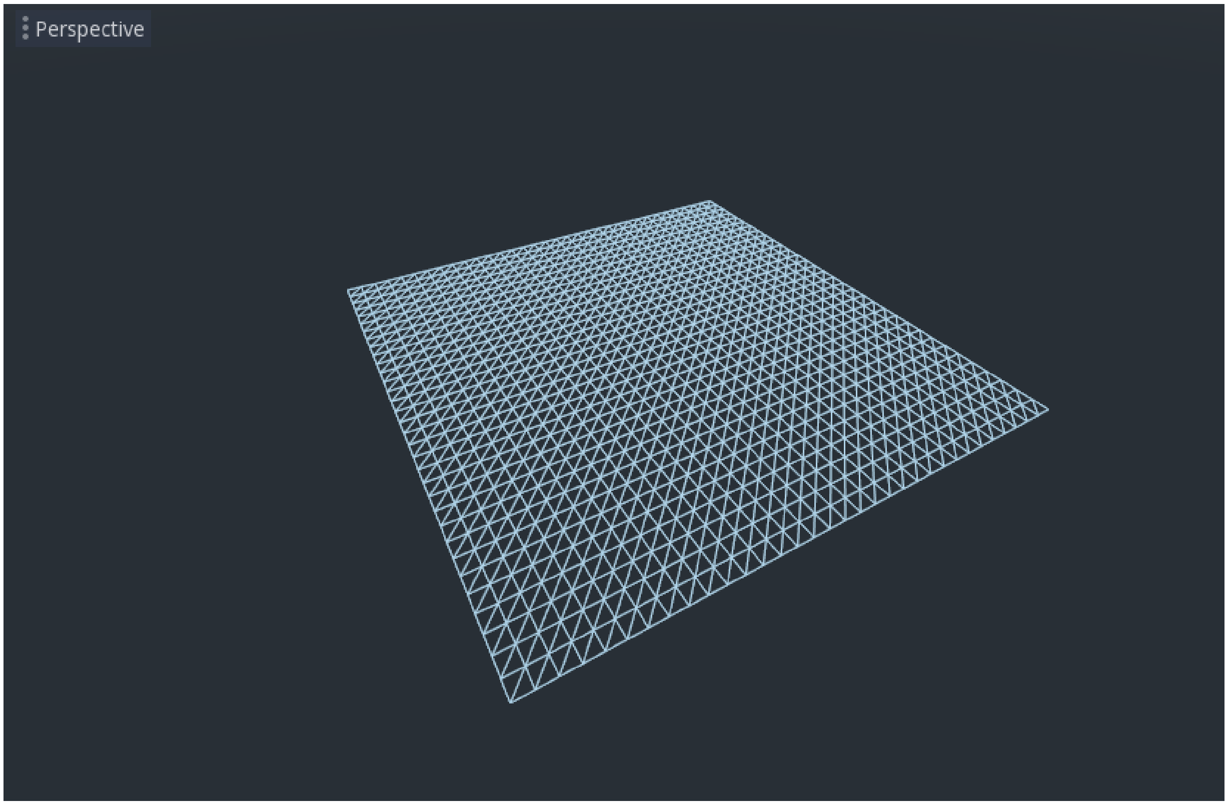

Inspector
Node
History
MeshInstance3D
Filter Properties
MeshInstance3D
Mesh

Size x 2 m, y 2 m

Subdivide Width 32

Subdivide Depth 32

You can see that there are now many more triangles in the MeshInstance3D. This will give us more vertices to work with and thus allow us to add more detail.

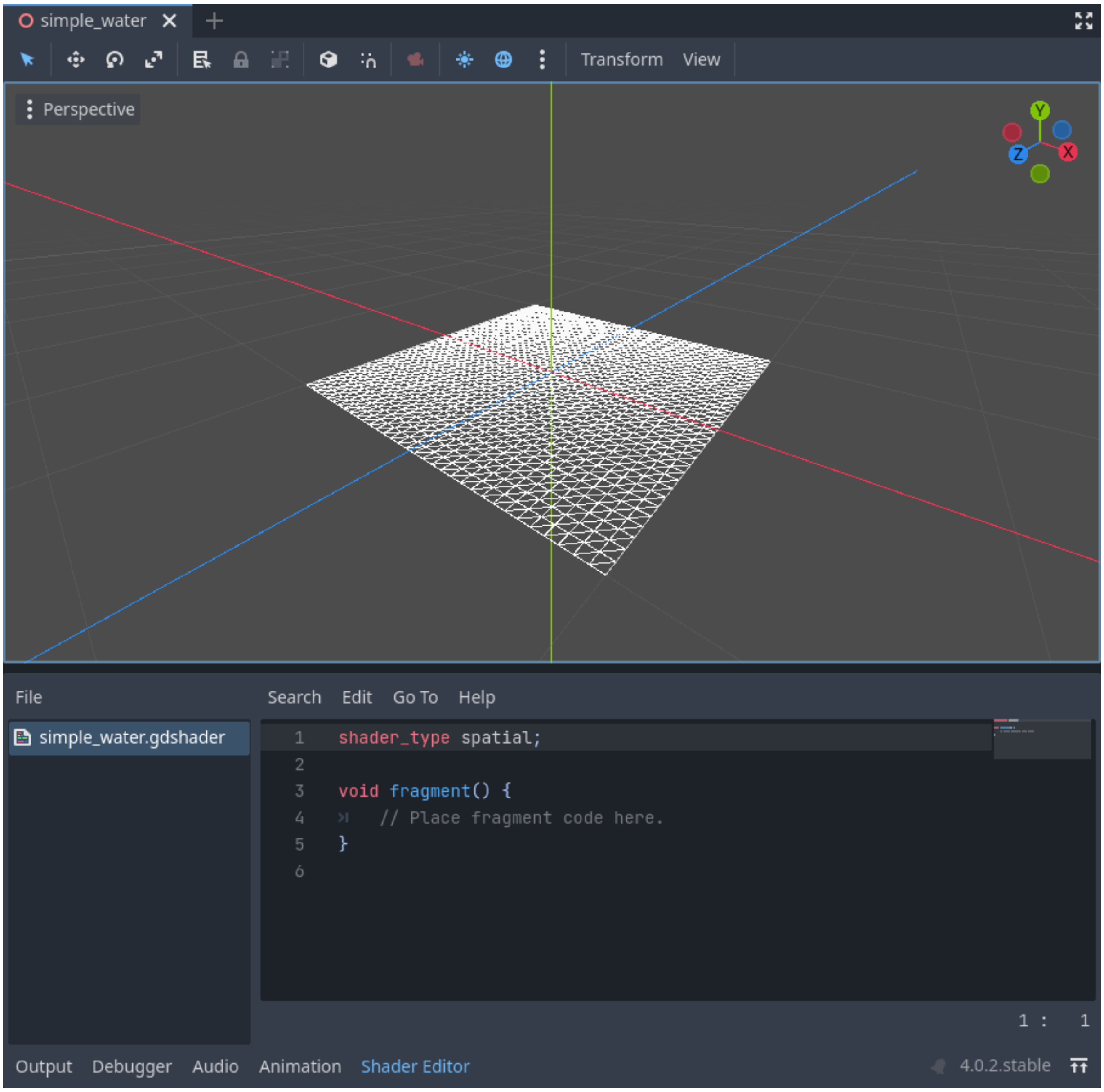


PrimitiveMeshes, like PlaneMesh, only have one surface, so instead of an array of materials there is only one. Click beside "Material" where it says "[empty]" and select "New ShaderMaterial". Then click the sphere that appears.

Now click beside "Shader" where it says "[empty]" and select "New Shader".

The shader editor should now pop up and you are ready to begin writing your first Spatial shader!

Shader magic


simple_water
Transform
View
Perspective
Y
X
Z
File
Search
Edit
Go To
Help
simple_water.gdshader
1 : 1
Output
Debugger
Audio
Animation
Shader Editor
4.0.2.stable

```
shader_type spatial;

void fragment() {
	// Place fragment code here.
}
```

The new shader is already generated with a `shader_type` variable and the `fragment()` function. The first thing Godot shaders need is a declaration of what type of shader they are. In this case the `shader_type` is set to `spatial` because this is a spatial shader.

```
shader_type spatial;
```

For now ignore the `fragment()` function and define the `vertex()` function. The `vertex()` function determines where the vertices of your MeshInstance3D appear in the final scene. We will be using it to offset the height of each vertex and make our flat plane appear like a little terrain.

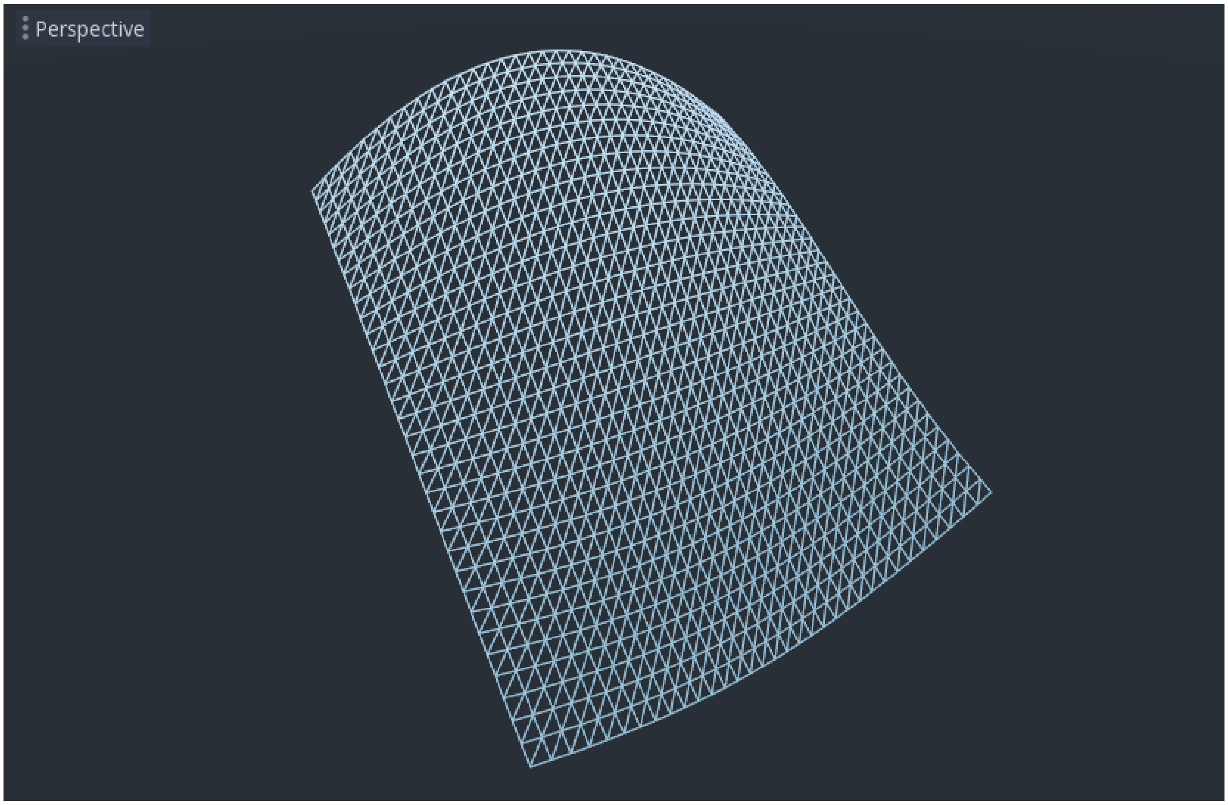
We define the vertex shader like so:

```
void vertex() {

}
```

With nothing in the `vertex()` function, Godot will use its default vertex shader. We can easily start to make changes by adding a single line:

```
void vertex() {
  VERTEX.y += cos(VERTEX.x) * sin(VERTEX.z);
}
```

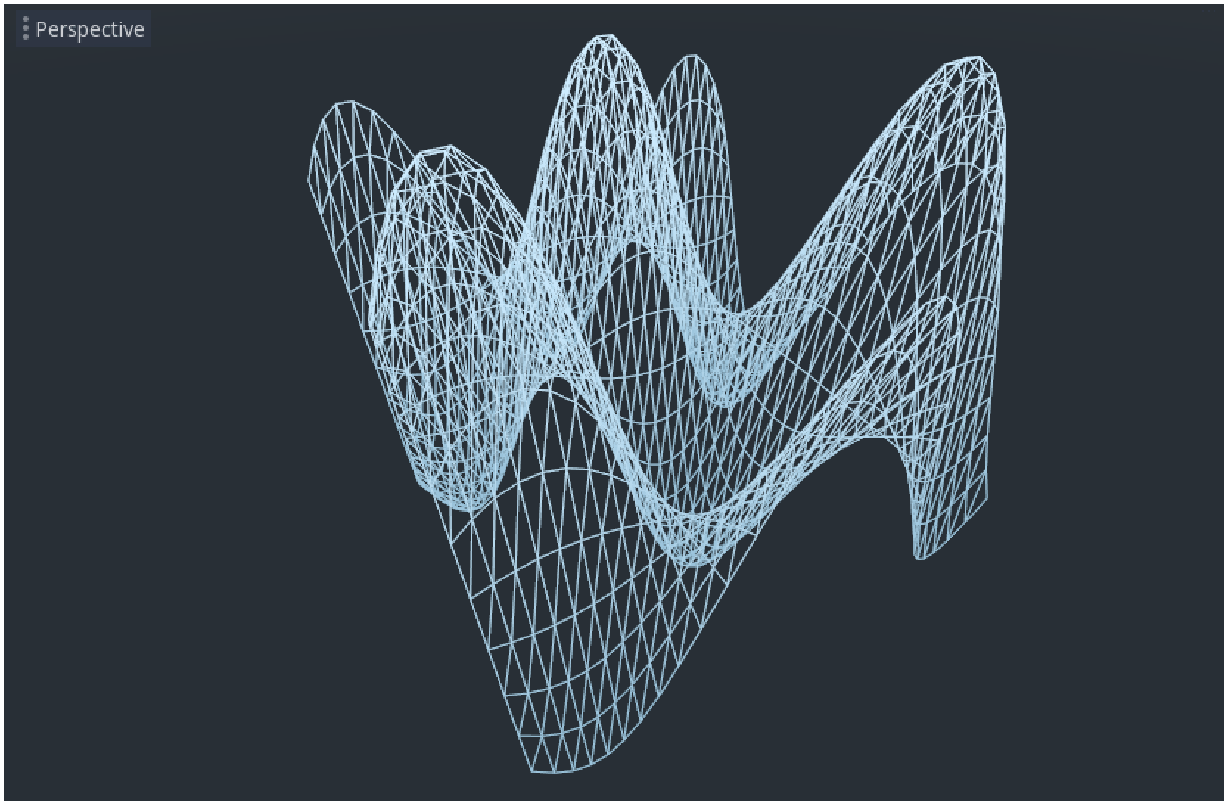
Adding this line, you should get an image like the one below.



Okay, let's unpack this. The `y` value of the `VERTEX` is being increased. And we are passing the `x` and `z` components of the `VERTEX` as arguments to `cos` and `sin`; that gives us a wave-like appearance across the `x` and `z` axes.

What we want to achieve is the look of little hills; after all. `cos` and `sin` already look kind of like hills. We do so by scaling the inputs to the `cos` and `sin` functions.

```
void vertex() {
  VERTEX.y += cos(VERTEX.x * 4.0) * sin(VERTEX.z * 4.0);
}
```



This looks better, but it is still too spiky and repetitive, let's make it a little more interesting.

Noise heightmap

Noise is a very popular tool for faking the look of terrain. Think of it as similar to the cosine function where you have repeating hills except, with noise, each hill has a different height.

Godot provides the NoiseTexture2D resource for generating a noise texture that can be accessed from a shader.

To access a texture in a shader add the following code near the top of your shader, outside the `vertex()` function.

```
uniform sampler2D noise;
```

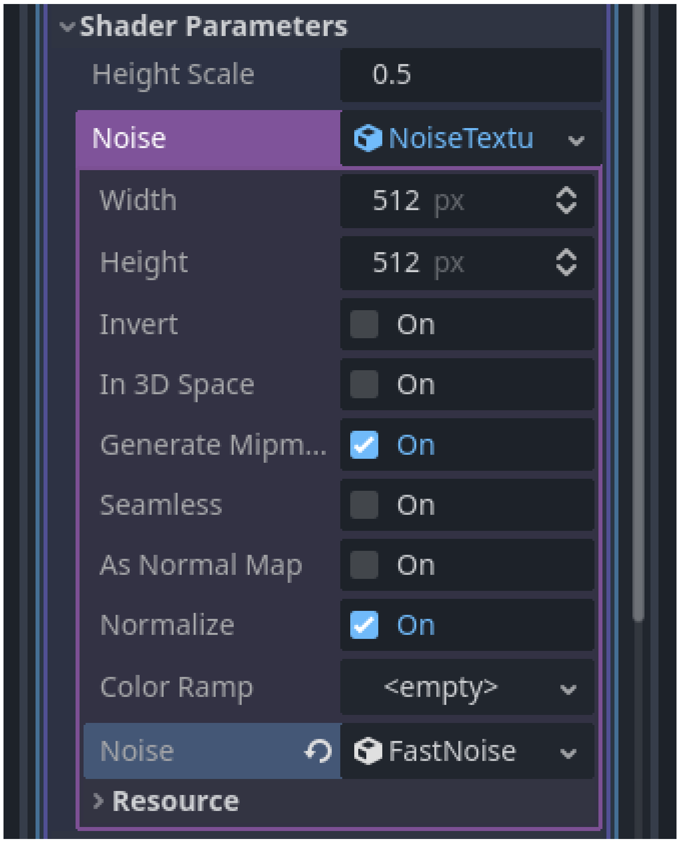
This will allow you to send a noise texture to the shader. Now look in the inspector under your material. You should see a section called "Shader Params". If you open it up, you'll see a section called "noise".

Click beside it where it says "[empty]" and select "New NoiseTexture2D". Then in your NoiseTexture2D click beside where it says "Noise" and select "New FastNoiseLite".

Note

FastNoiseLite is used by the NoiseTexture2D to generate a heightmap.

Once you set it up and should look like this.



Shader Parameters

Height Scale	0.5
Noise	NoiseTextu
Width	512 px
Height	512 px
Invert	☐ On
In 3D Space	☐ On
Generate Mipm...	☑ On
Seamless	☐ On
As Normal Map	☐ On
Normalize	☑ On
Color Ramp	<empty>
Noise	FastNoise

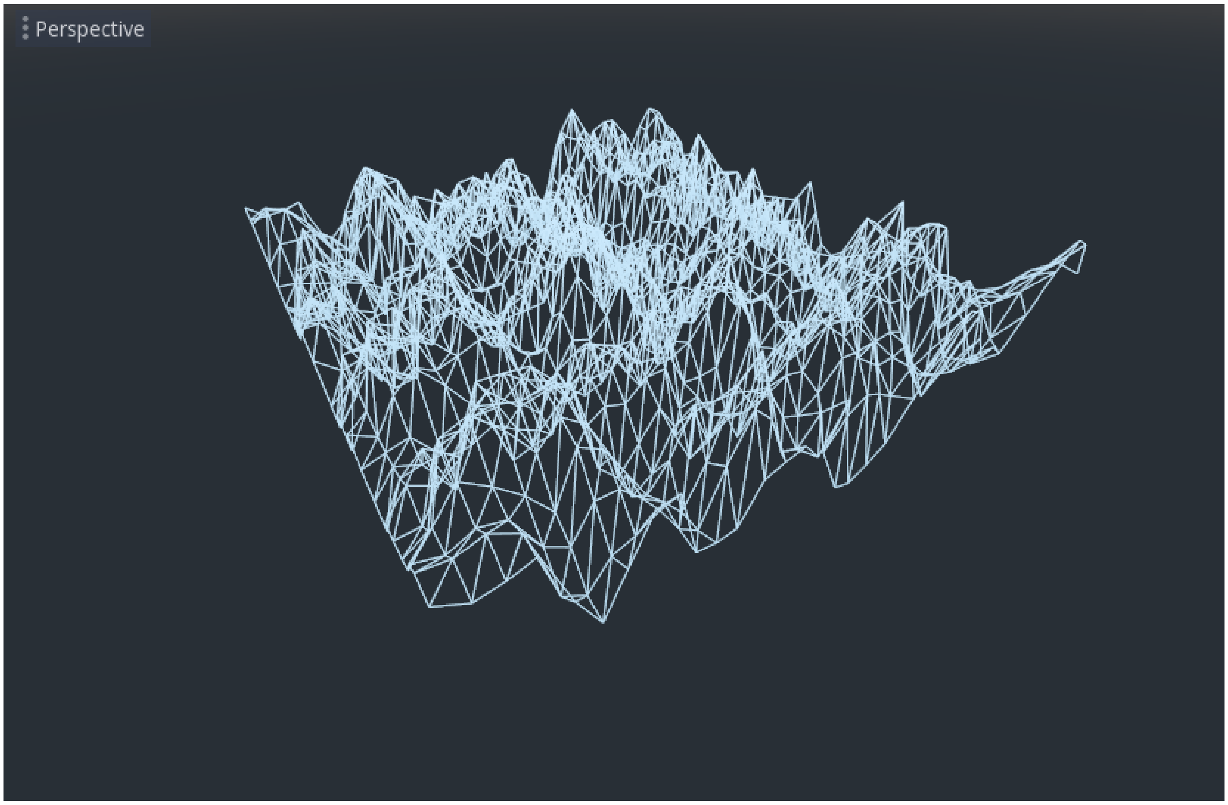
Resource

Now, access the noise texture using the `texture()` function. `texture()` takes a texture as the first argument and a `vec2` for the position on the texture as the second argument. We use the `x` and `z` channels of `VERTEX` to determine where on the texture to look up. Note that the PlaneMesh coordinates are within the [-1,1] range (for a size of 2), while the texture coordinates are within [0,1], so to normalize we divide by the size of the PlaneMesh by 2.0 and add 0.5. `texture()` returns a `vec4` of the `r, g, b, a` channels at the position. Since the noise texture is grayscale, all of the values are the same, so we can use any one of the channels as the height. In this case we'll use the `r`, or `x` channel.

```
void vertex() {
  float height = texture(noise, VERTEX.xz / 2.0 + 0.5).x;
  VERTEX.y += height;
}
```

Note: `xyzw` is the same as `rgba` in GLSL, so instead of `texture().x` above, we could use `texture().r`. See the OpenGL documentation [https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)#Vectors] for more details.

Using this code you can see the texture creates random looking hills.



Right now it is too spiky, we want to soften the hills a bit. To do that, we will use a uniform. You already used a uniform above to pass in the noise texture, now let's learn how they work.

Uniforms

Uniform variables allow you to pass data from the game into the shader. They are very useful for controlling shader effects. Uniforms can be almost any datatype that can be used in the shader. To use a uniform, you declare it in your Shader using the keyword `uniform`.

Let's make a uniform that changes the height of the terrain.

```
uniform float height_scale = 0.5;
```

Godot lets you initialize a uniform with a value; here, `height_scale` is set to `0.5`. You can set uniforms from GDScript by calling the function `set_shader_parameter()` on the material corresponding to the shader. The value passed from GDScript takes precedence over the value used to initialize it in the shader.

```
# called from the MeshInstance3D
mesh.material.set_shader_parameter("height_scale", 0.5)
```

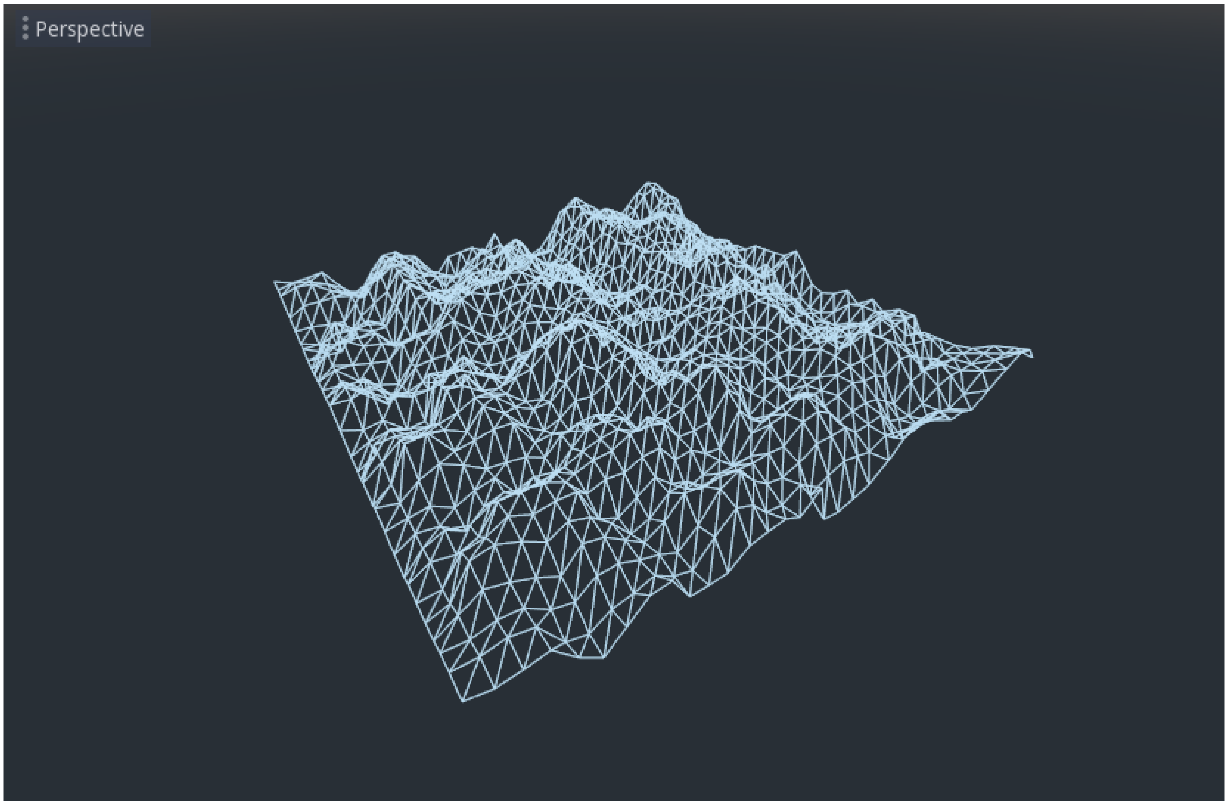
Note

Changing uniforms in Spatial-based nodes is different from CanvasItem-based nodes. Here, we set the material inside the PlaneMesh resource. In other mesh resources you may need to first access the material by calling `surface_get_material()`. While in the MeshInstance3D you would access the material using `get_surface_material()` or `material_override`.

Remember that the string passed into `set_shader_parameter()` must match the name of the uniform variable in the Shader. You can use the uniform variable anywhere inside your Shader. Here, we will use it to set the height value instead of arbitrarily multiplying by `0.5`.

```
VERTEX.y += height * height_scale;
```

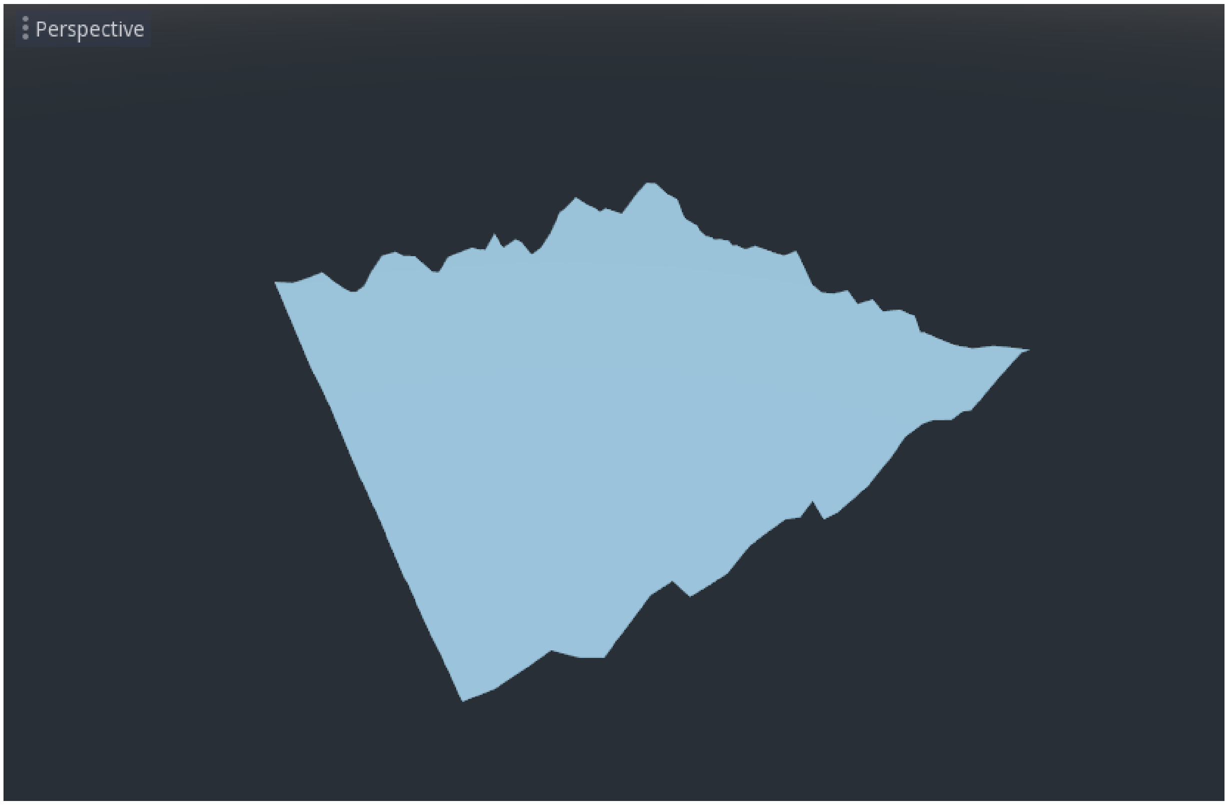
Now it looks much better.



Using uniforms, we can even change the value every frame to animate the height of the terrain. Combined with Tweens, this can be especially useful for animations.

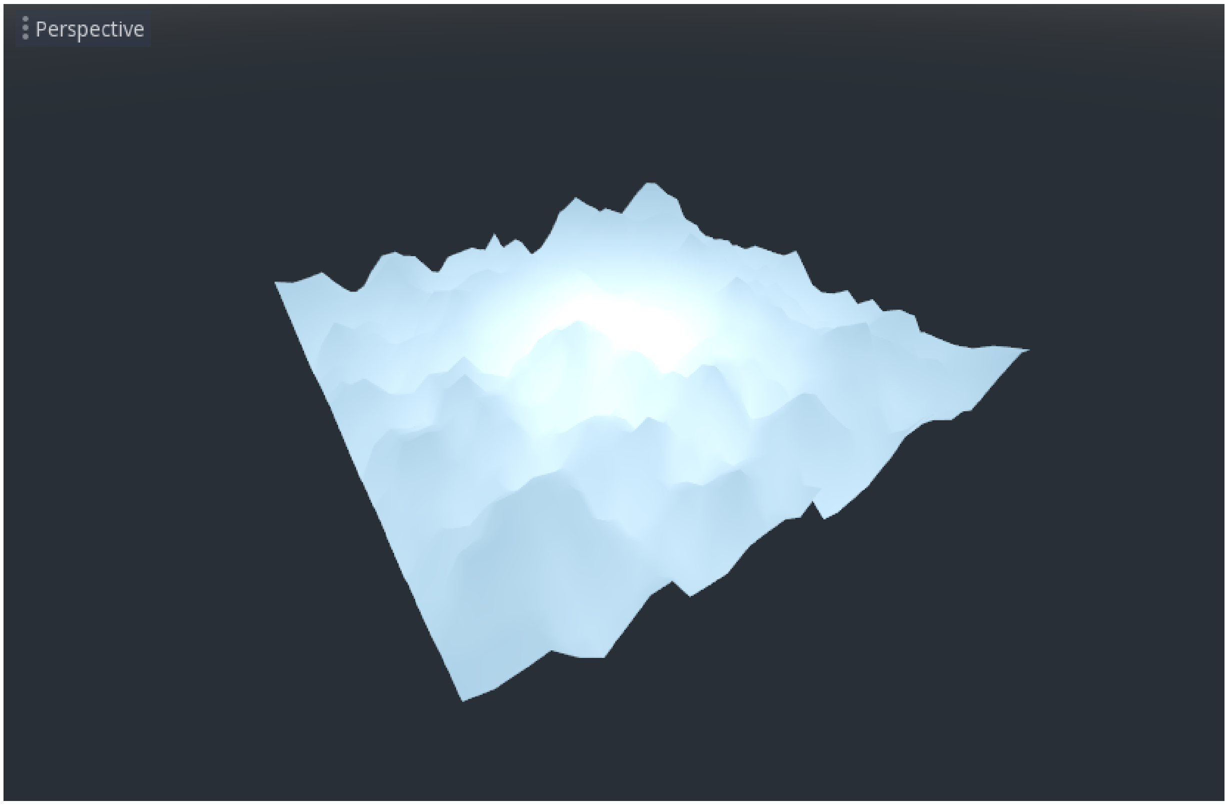
Interacting with light

First, turn wireframe off. To do so, click in the upper-left of the Viewport again, where it says "Perspective", and select "Display Normal". Additionally in the 3D scene toolbar, turn off preview sunlight.



Note how the mesh color goes flat. This is because the lighting on it is flat. Let's add a light!

First, we will add an OmniLight3D to the scene.



You can see the light affecting the terrain, but it looks odd. The problem is the light is affecting the terrain as if it were a flat plane. This is because the light shader uses the normals from the Mesh to calculate light.

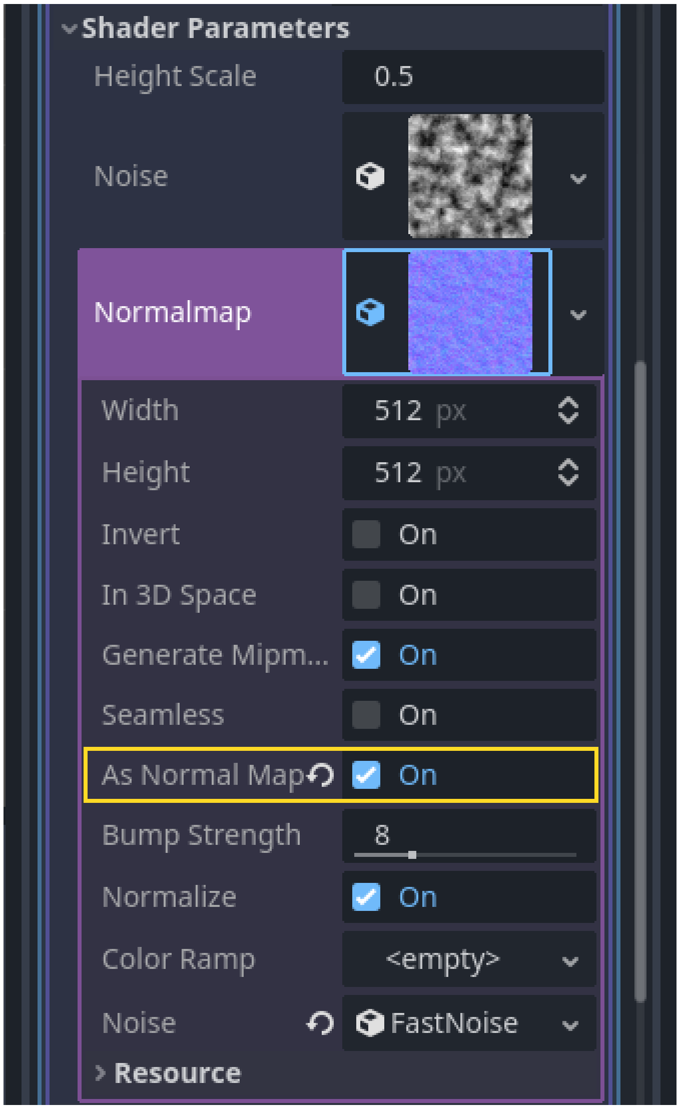
The normals are stored in the Mesh, but we are changing the shape of the Mesh in the shader, so the normals are no longer correct. To fix this, we can recalculate the normals in the shader or use a normal texture that corresponds to our noise. Godot makes both easy for us.

You can calculate the new normal manually in the vertex function and then just set `NORMAL`. With `NORMAL` set, Godot will do all the difficult lighting calculations for us. We will cover this method in the next part of this tutorial, for now we will read normals from a texture.

Instead we will rely on the NoiseTexture again to calculate normals for us. We do that by passing in a second noise texture.

```
uniform sampler2D normalmap;
```

Set this second uniform texture to another NoiseTexture2D with another FastNoiseLite. But this time, check **As Normalmap**.



Shader Parameters

Height Scale	0.5
Noise	
Normalmap	
Width	512 px
Height	512 px
Invert	☐ On
In 3D Space	☐ On
Generate Mipm...	☑ On
Seamless	☐ On
As Normal Map	☑ On
Bump Strength	8
Normalize	☑ On
Color Ramp	<empty>
Noise	FastNoise

Resource

Now, because this is a normalmap and not a per-vertex normal, we are going to assign it in the `fragment()` function. The `fragment()` function will be explained in more detail in the next part of this tutorial.

```
void fragment() {
}
```

When we have normals that correspond to a specific vertex we set `NORMAL`, but if you have a normalmap that comes from a texture, set the normal using `NORMAL_MAP`. This way Godot will handle the wrapping of texture around the mesh automatically.

Lastly, in order to ensure that we are reading from the same places on the noise texture and the normalmap texture, we are going to pass the `VERTEX.xz` position from the `vertex()` function to the `fragment()` function. We do that with varyings.

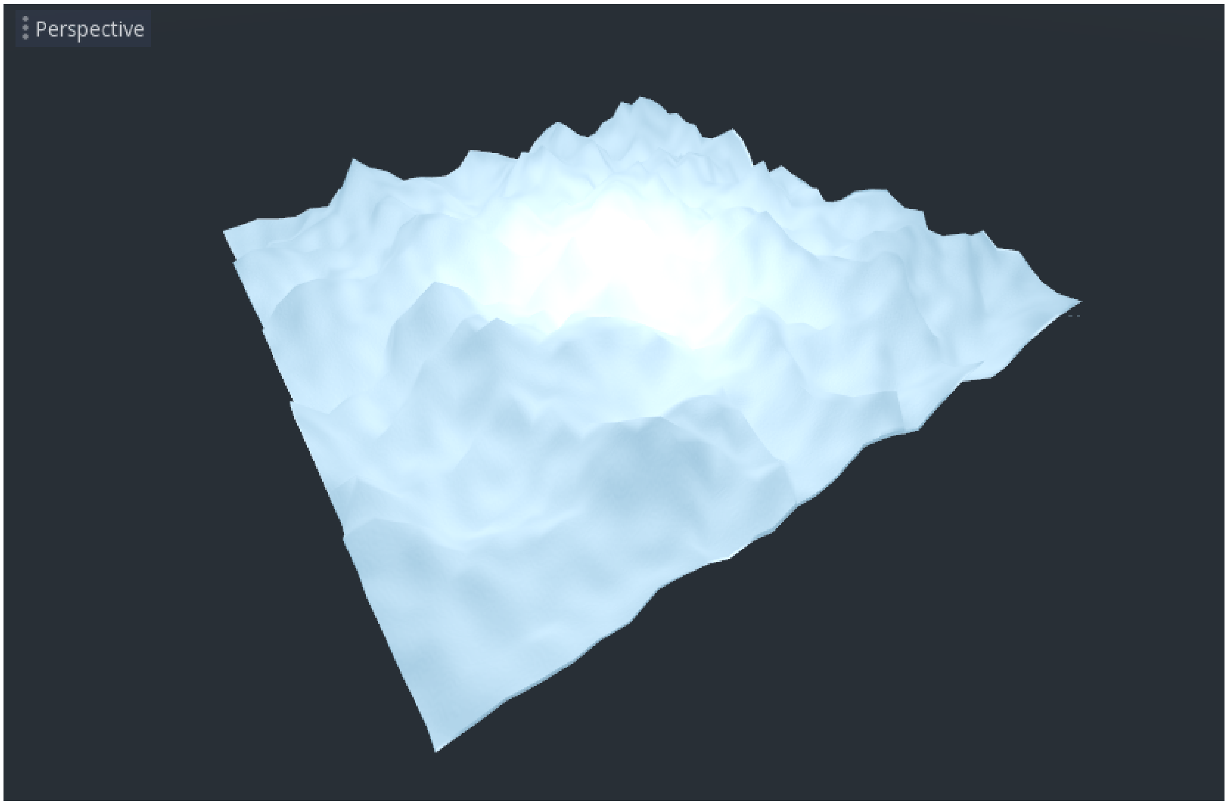
Above the `vertex()` define a `vec2` called `tex_position`. And inside the `vertex()` function assign `VERTEX.xz` to `tex_position`.

```
varying vec2 tex_position;

void vertex() {
  ...
  tex_position = VERTEX.xz / 2.0 + 0.5;
  float height = texture(noise, tex_position).x;
  ...
}
```

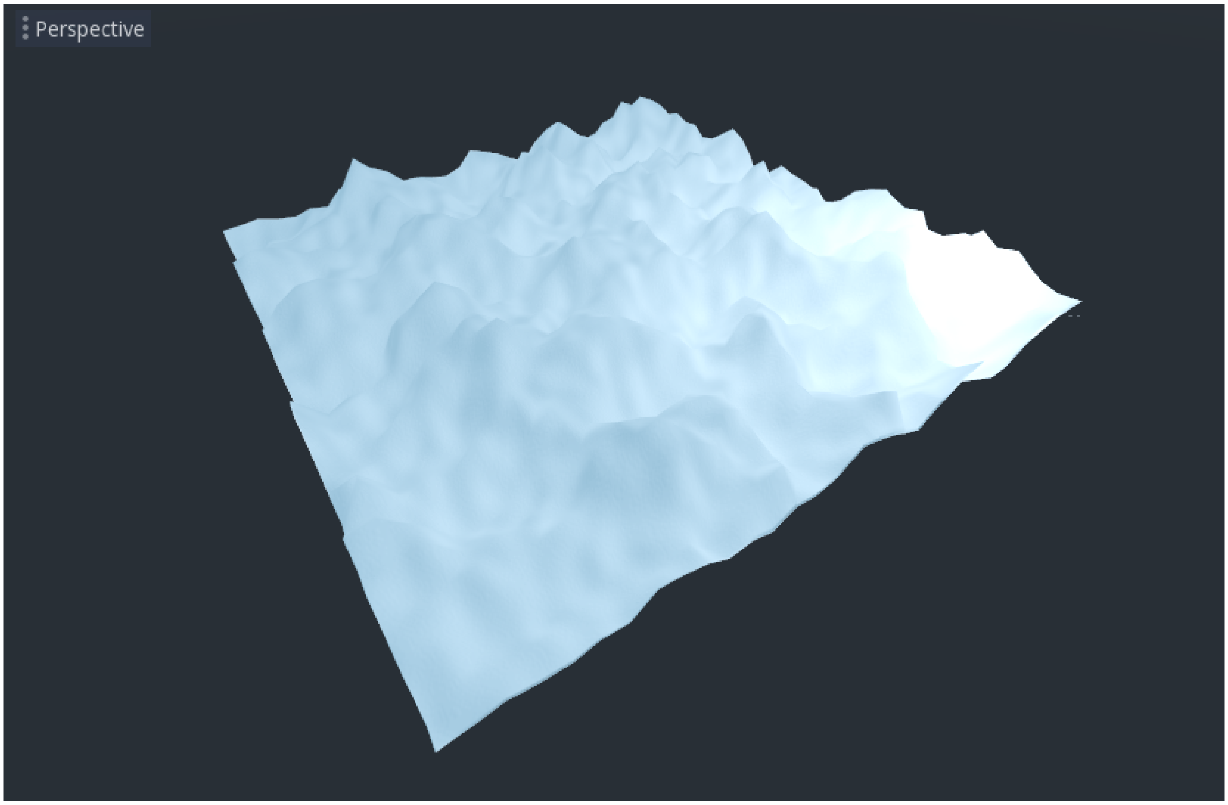
And now we can access `tex_position` from the `fragment()` function.

```
void fragment() {
  NORMAL_MAP = texture(normalmap, tex_position).xyz;
}
```

With the normals in place the light now reacts to the height of the mesh dynamically.


Perspective

We can even drag the light around and the lighting will update automatically.



Here is the full code for this tutorial. You can see it is not very long as Godot handles most of the difficult stuff for you.

```
shader_type spatial;

uniform float height_scale = 0.5;
uniform sampler2D noise;
uniform sampler2D normalmap;

varying vec2 tex_position;

void vertex() {
  tex_position = VERTEX.xz / 2.0 + 0.5;
  float height = texture(noise, tex_position).x;
  VERTEX.y += height * height_scale;
}

void fragment() {
  NORMAL_MAP = texture(normalmap, tex_position).xyz;
}
```

That is everything for this part. Hopefully, you now understand the basics of vertex shaders in Godot. In the next part of this tutorial we will write a fragment function to accompany this vertex function and we will cover a more advanced technique to turn this terrain into an ocean of moving waves.

Your second 3D shader

From a high-level, what Godot does is give the user a bunch of parameters that can be optionally set (`AO`, `SSS_Strength`, `RIM`, etc.). These parameters correspond to different complex effects (Ambient Occlusion, SubSurface Scattering, Rim Lighting, etc.). When not written to, the code is thrown out before it is compiled and so the shader does not incur the cost of the extra feature. This makes it easy for users to have complex PBR-correct shading, without writing complex shaders. Of course, Godot also allows you to ignore all these parameters and write a fully customized shader.

For a full list of these parameters see the spatial shader reference doc.

A difference between the vertex function and a fragment function is that the vertex function runs per vertex and sets properties such as `VERTEX` (position) and `NORMAL`, while the fragment shader runs per pixel and, most importantly, sets the `ALBEDO` color of the MeshInstance3D.

Your first spatial fragment function

As mentioned in the previous part of this tutorial. The standard use of the fragment function in Godot is to set up different material properties and let Godot handle the rest. In order to provide even more flexibility, Godot also provides things called render modes. Render modes are set at the top of the shader, directly below `shader_type`, and they specify what sort of functionality you want the built-in aspects of the shader to have.

For example, if you do not want to have lights affect an object, set the render mode to `unshaded`:

```
render_mode unshaded;
```

You can also stack multiple render modes together. For example, if you want to use toon shading instead of more-realistic PBR shading, set the diffuse mode and specular mode to toon:

```
render_mode diffuse_toon, specular_toon;
```

This model of built-in functionality allows you to write complex custom shaders by changing only a few parameters.

For a full list of render modes see the Spatial shader reference.

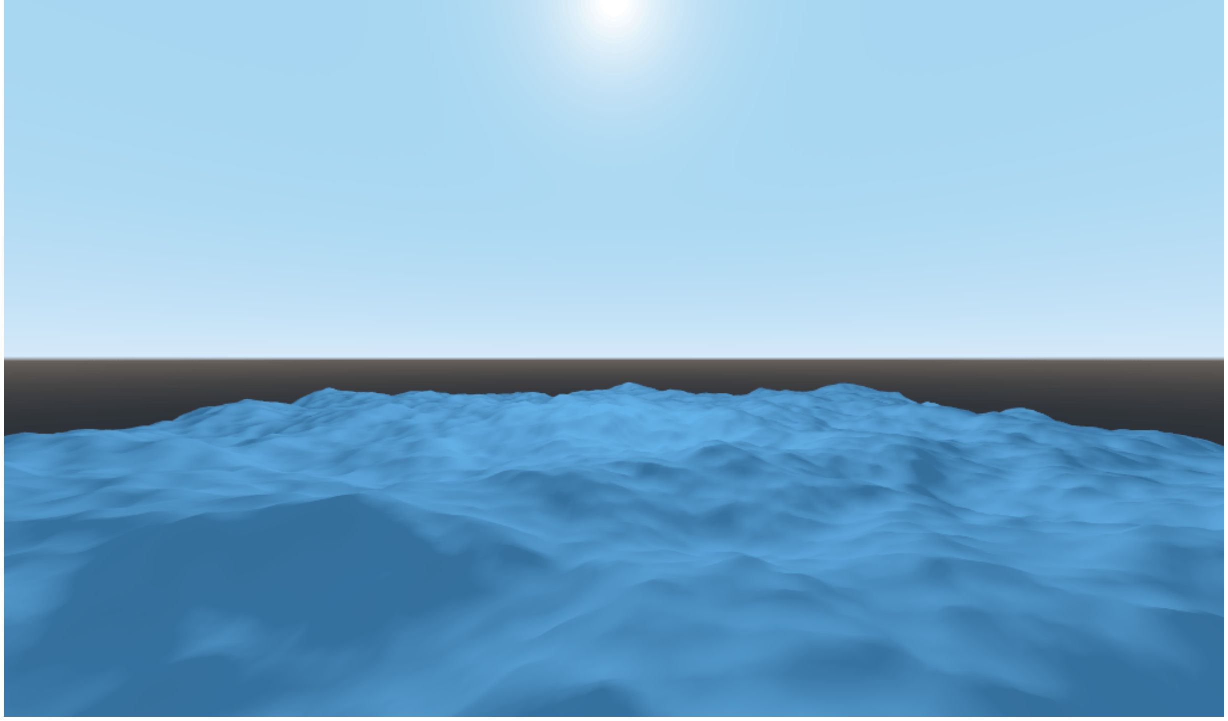
In this part of the tutorial, we will walk through how to take the bumpy terrain from the previous part and turn it into an ocean.

First let's set the color of the water. We do that by setting `ALBEDO`.

`ALBEDO` is a `vec3` that contains the color of the object.

Let's set it to a nice shade of blue.

```
void fragment() {
  ALBEDO = vec3(0.1, 0.3, 0.5);
}
```



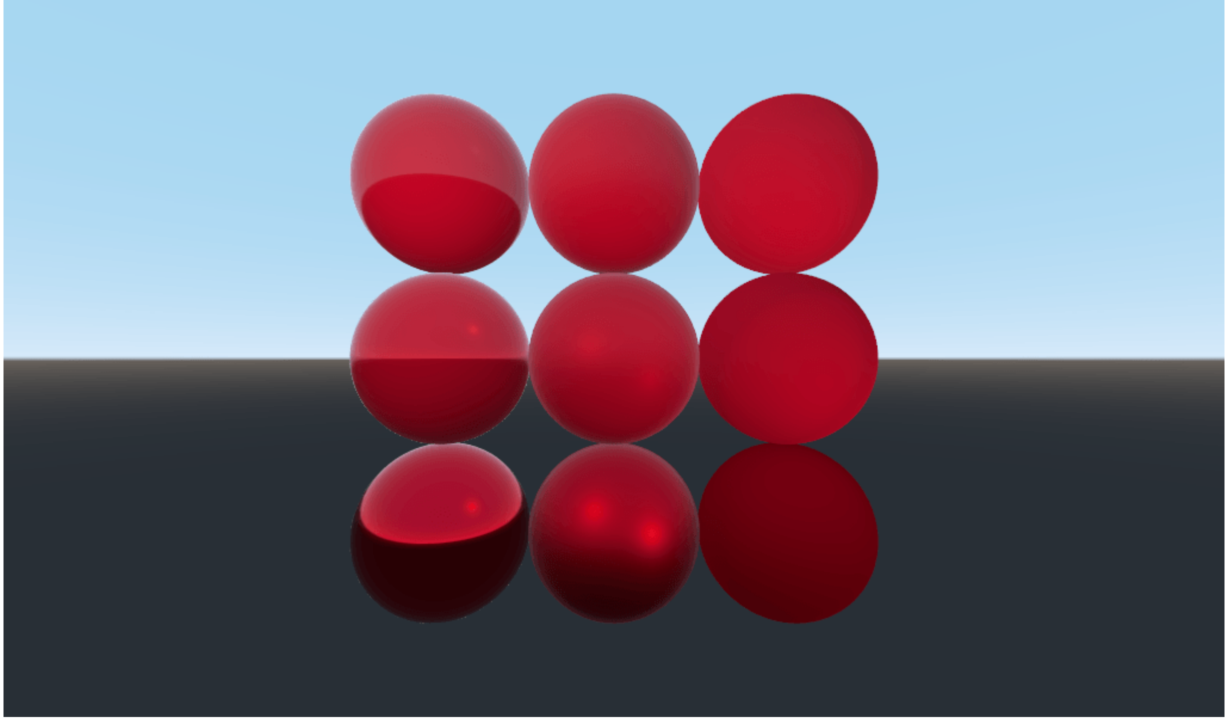
We set it to a very dark shade of blue because most of the blueness of the water will come from reflections from the sky.

The PBR model that Godot uses relies on two main parameters: `METALLIC` and `ROUGHNESS`.

`ROUGHNESS` specifies how smooth/rough the surface of a material is. A low `ROUGHNESS` will make a material appear like a shiny plastic, while a high roughness makes the material appear more solid in color.

`METALLIC` specifies how much like a metal the object is. It is better set close to `0` or `1`. Think of `METALLIC` as changing the balance between the reflection and the `ALBEDO` color. A high `METALLIC` almost ignores `ALBEDO` altogether, and looks like a mirror of the sky. While a low `METALLIC` has a more equal representation of sky color and `ALBEDO` color.

`ROUGHNESS` increases from `0` to `1` from left to right while `METALLIC` increase from `0` to `1` from top to bottom.

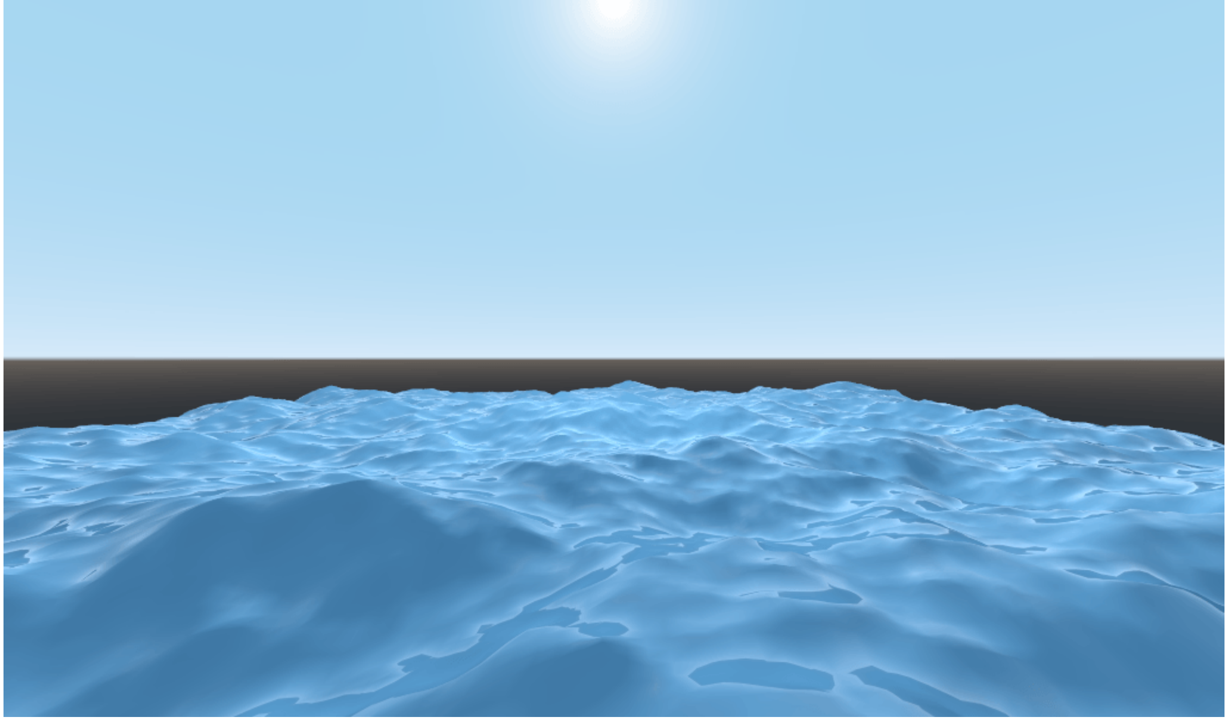


Note

`METALLIC` should be close to `0` or `1` for proper PBR shading. Only set it between them for blending between materials.

Water is not a metal, so we will set its `METALLIC` property to `0.0`. Water is also highly reflective, so we will set its `ROUGHNESS` property to be quite low as well.

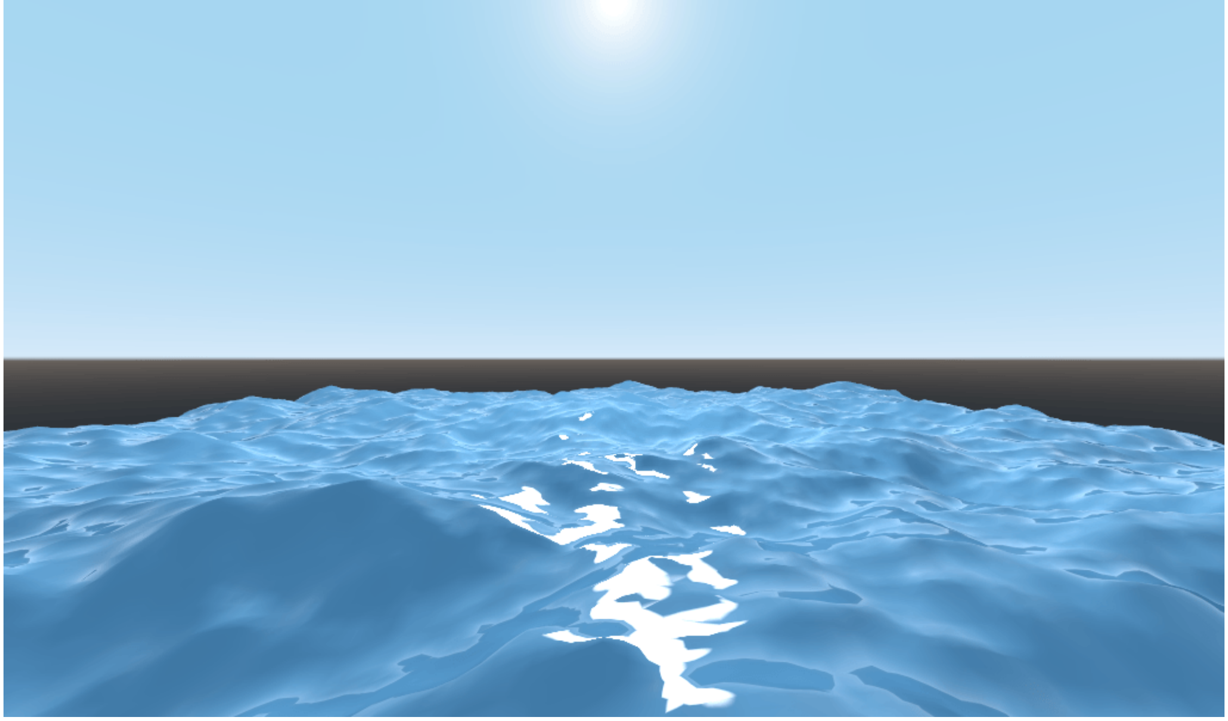
```
void fragment() {
  METALLIC = 0.0;
  ROUGHNESS = 0.01;
  ALBEDO = vec3(0.1, 0.3, 0.5);
}
```



Now we have a smooth plastic looking surface. It is time to think about some particular properties of water that we want to emulate. There are two main ones that will take this from a weird plastic surface to nice stylized water. The first is specular reflections. Specular reflections are those bright spots you see from where the sun reflects directly into your eye. The second is fresnel reflectance. Fresnel reflectance is the property of objects to become more reflective at shallow angles. It is the reason why you can see into water below you, but farther away it reflects the sky.

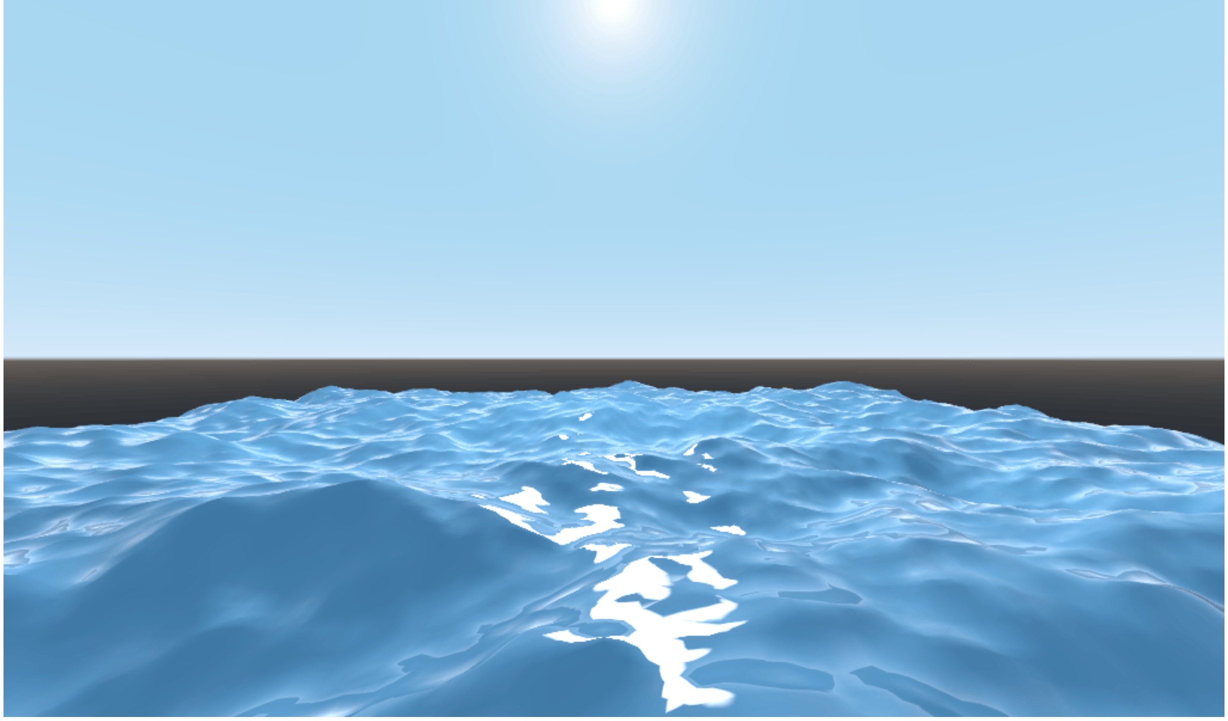
In order to increase the specular reflections, we will do two things. First, we will change the render mode for specular to toon because the toon render mode has larger specular highlights.

```
render_mode specular_toon;
```



Second we will add rim lighting. Rim lighting increases the effect of light at glancing angles. Usually it is used to emulate the way light passes through fabric on the edges of an object, but we will use it here to help achieve a nice watery effect.

```
void fragment() {
  RIM = 0.2;
  METALLIC = 0.0;
  ROUGHNESS = 0.01;
  ALBEDO = vec3(0.1, 0.3, 0.5);
}
```

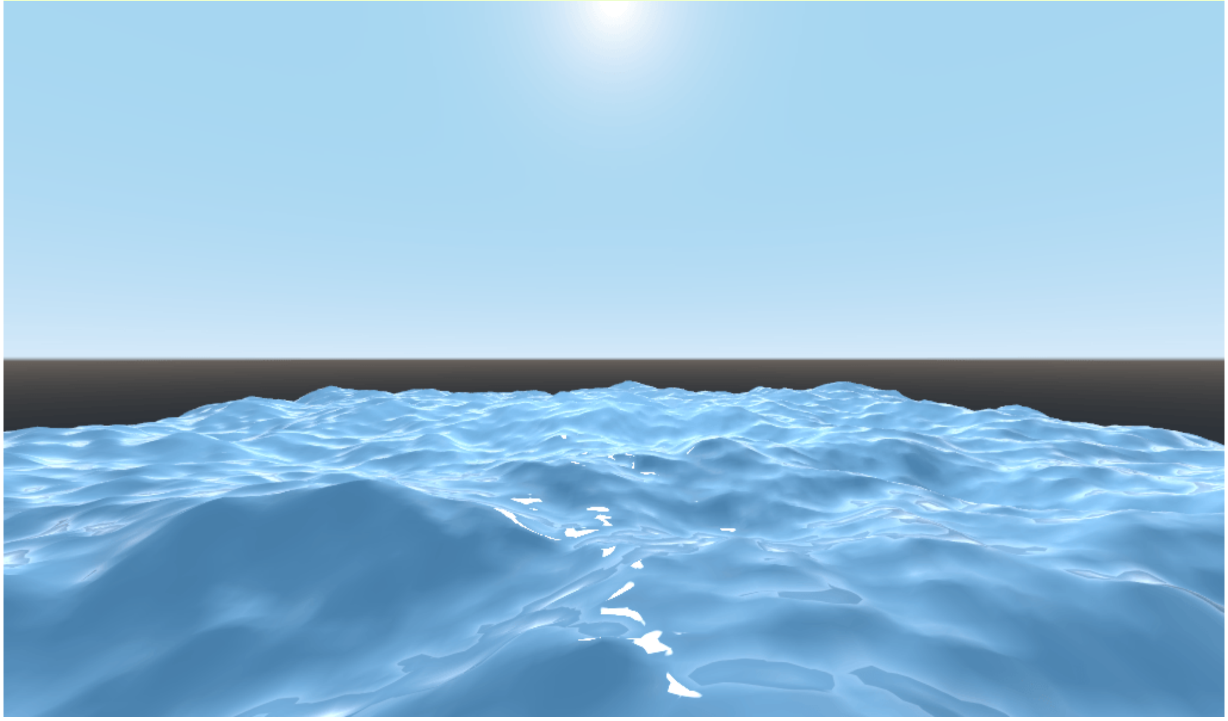


In order to add fresnel reflectance, we will compute a fresnel term in our fragment shader. Here, we aren't going to use a real fresnel term for performance reasons. Instead, we'll approximate it using the dot product of the `NORMAL` and `VIEW` vectors. The `NORMAL` vector points away from the mesh's surface, while the `VIEW` vector is the direction between your eye and that point on the surface. The dot product between them is a handy way to tell when you are looking at the surface head-on or at a glancing angle.

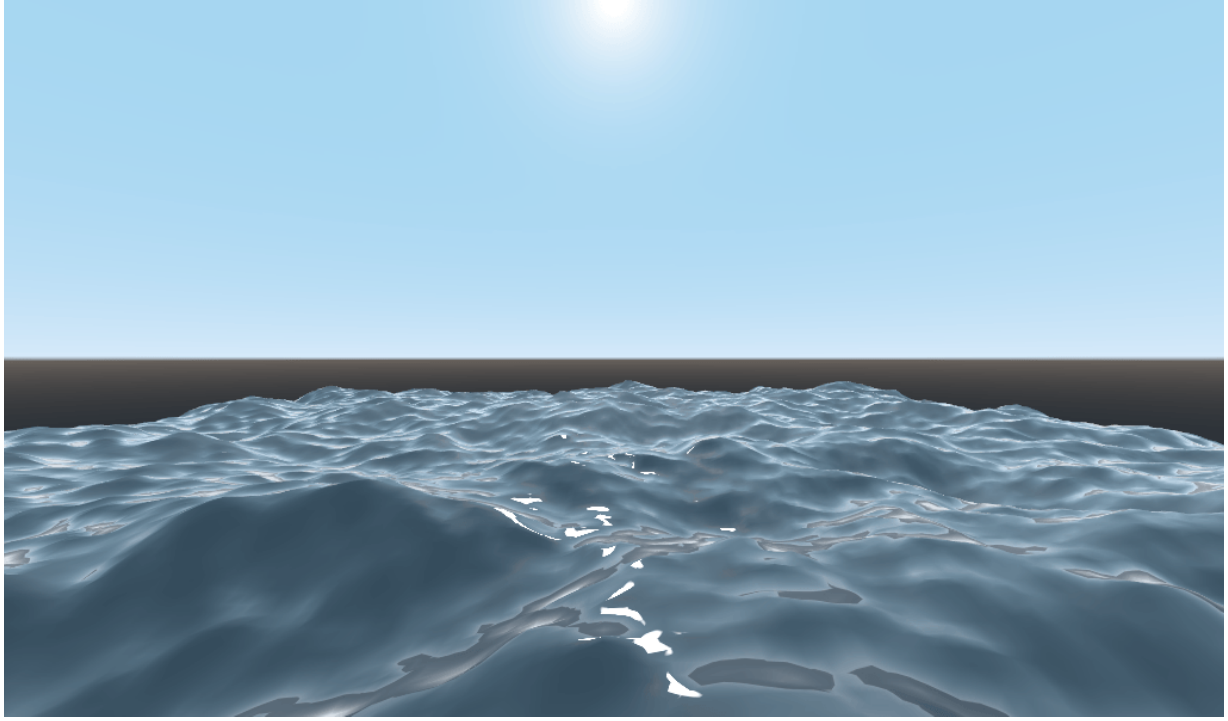
```
float fresnel = sqrt(1.0 - dot(NORMAL, VIEW));
```

And mix it into both `ROUGHNESS` and `ALBEDO`. This is the benefit of ShaderMaterials over StandardMaterial3Ds. With StandardMaterial3D, we could set these properties with a texture, or to a flat number. But with shaders we can set them based on any mathematical function that we can dream up.

```
void fragment() {
  float fresnel = sqrt(1.0 - dot(NORMAL, VIEW));
  RIM = 0.2;
  METALLIC = 0.0;
  ROUGHNESS = 0.01 * (1.0 - fresnel);
  ALBEDO = vec3(0.1, 0.3, 0.5) + (0.1 * fresnel);
}
```



And now, with only 5 lines of code, you can have complex looking water. Now that we have lighting, this water is looking too bright. Let's darken it. This is done easily by decreasing the values of the `vec3` we pass into `ALBEDO`. Let's set them to `vec3(0.01, 0.03, 0.05)`.



Animating with `TIME`

Going back to the vertex function, we can animate the waves using the built-in variable `TIME`.

`TIME` is a built-in variable that is accessible from the vertex and fragment functions.

In the last tutorial we calculated height by reading from a heightmap. For this tutorial, we will do the same. Put the heightmap code in a function called `height()`.

```
float height(vec2 position) {
  return texture(noise, position / 10.0).x; // Scaling factor
is based on mesh size (this PlaneMesh is 10×10).
}
```

In order to use `TIME` in the `height()` function, we need to pass it in.

```
float height(vec2 position, float time) {
}
```

And make sure to correctly pass it in inside the vertex function.

```
void vertex() {
  vec2 pos = VERTEX.xz;
  float k = height(pos, TIME);
  VERTEX.y = k;
}
```

Instead of using a normalmap to calculate normals. We are going to compute them manually in the `vertex()` function. To do so use the following line of code.

```
NORMAL = normalize(vec3(k - height(pos + vec2(0.1, 0.0), TIME),
0.1, k - height(pos + vec2(0.0, 0.1), TIME)));
```

We need to compute `NORMAL` manually because in the next section we will be using math to create complex-looking waves.

Now, we are going to make the `height()` function a little more complicated by offsetting `position` by the cosine of `TIME`.

```
float height(vec2 position, float time) {
  vec2 offset = 0.01 * cos(position + time);
  return texture(noise, (position / 10.0) - offset).x;
}
```

This results in waves that move slowly, but not in a very natural way. The next section will dig deeper into using shaders to create more complex effects, in this case realistic waves, by adding a few more mathematical functions.

Advanced effects: waves

What makes shaders so powerful is that you can achieve complex effects by using math. To illustrate this, we are going to take our waves to the next level by modifying the `height()` function and by introducing a new function called `wave()`.

`wave()` has one parameter, `position`, which is the same as it is in `height()`.

We are going to call `wave()` multiple times in `height()` in order to fake the way waves look.

```
float wave(vec2 position){
  position += texture(noise, position / 10.0).x * 2.0 - 1.0;
  vec2 wv = 1.0 - abs(sin(position));
  return pow(1.0 - pow(wv.x * wv.y, 0.65), 4.0);
}
```

At first this looks complicated. So let's go through it line-by-line.

```
position += texture(noise, position / 10.0).x * 2.0 - 1.0;
```

Offset the position by the `noise` texture. This will make the waves curve, so they won't be straight lines completely aligned with the grid.

```
vec2 wv = 1.0 - abs(sin(position));
```

Define a wave-like function using `sin()` and `position`. Normally `sin()` waves are very round. We use `abs()` to absolute to give them a sharp ridge and constrain them to the 0-1 range. And then we subtract it from `1.0` to put the peak on top.

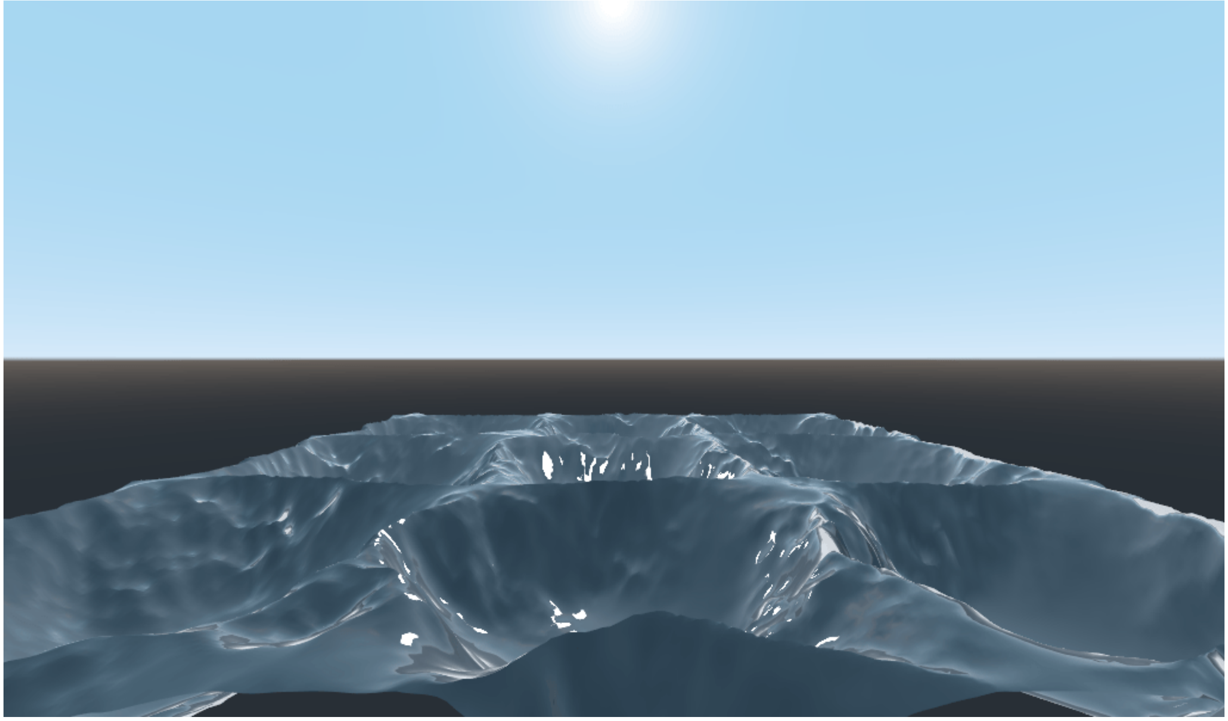
```
return pow(1.0 - pow(wv.x * wv.y, 0.65), 4.0);
```

Multiply the x-directional wave by the y-directional wave and raise it to a power to sharpen the peaks. Then subtract that from `1.0` so that the ridges become peaks and raise that to a power to sharpen the ridges.

We can now replace the contents of our `height()` function with `wave()`.

```
float height(vec2 position, float time) {
  float h = wave(position);
  return h;
}
```

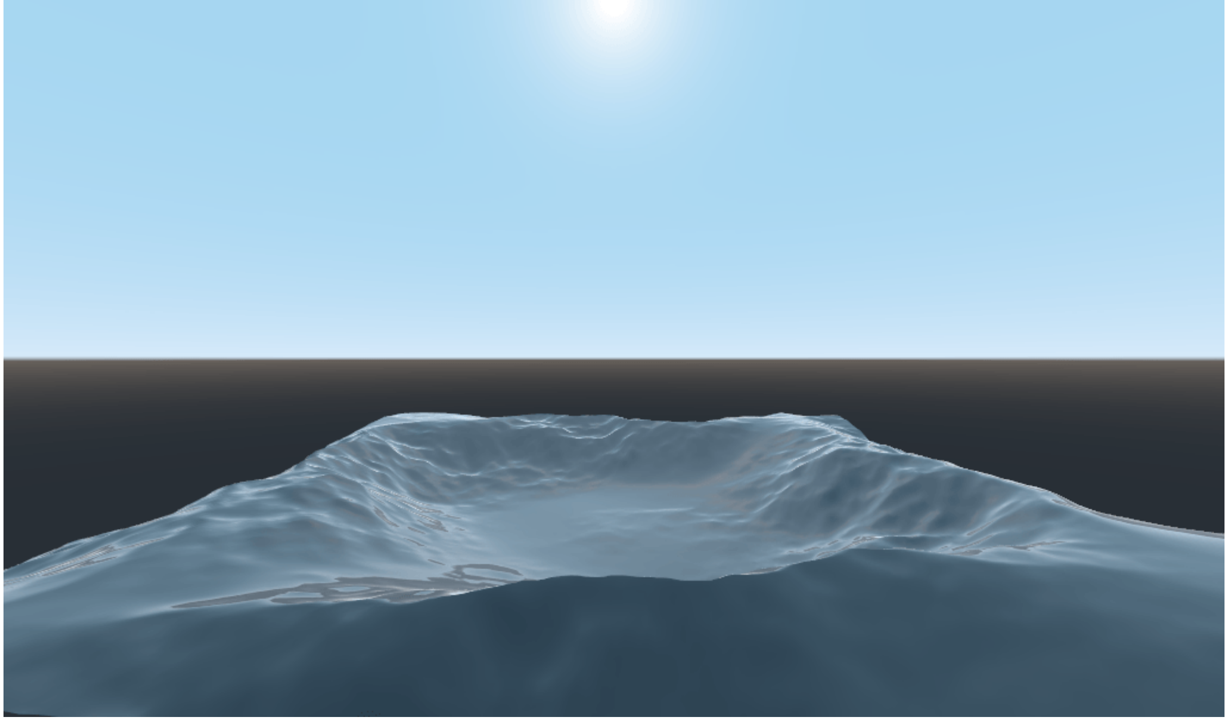
Using this, you get:



The shape of the sin wave is too obvious. So let's spread the waves out a bit. We do this by scaling `position`.

```
float height(vec2 position, float time) {
  float h = wave(position * 0.4);
  return h;
}
```

Now it looks much better.



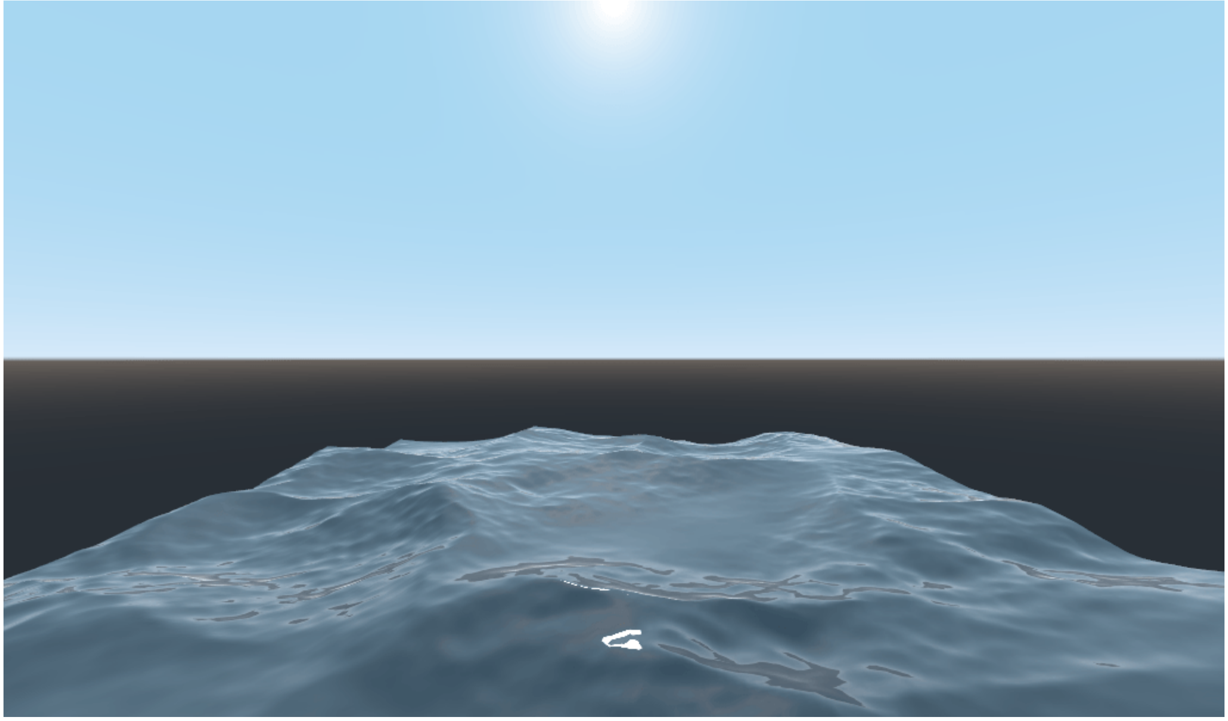
We can do even better if we layer multiple waves on top of each other at varying frequencies and amplitudes. What this means is that we are going to scale position for each one to make the waves thinner or wider (frequency). And we are going to multiply the output of the wave to make them shorter or taller (amplitude).

Here is an example for how you could layer the four waves to achieve nicer looking waves.

```
float height(vec2 position, float time) {
  float d = wave((position + time) * 0.4) * 0.3;
  d += wave((position - time) * 0.3) * 0.3;
  d += wave((position + time) * 0.5) * 0.2;
  d += wave((position - time) * 0.6) * 0.2;
  return d;
}
```

Note that we add time to two and subtract it from the other two. This makes the waves move in different directions creating a complex effect. Also note that the amplitudes (the number the result is multiplied by) all add up to `1.0`. This keeps the wave in the 0-1 range.

With this code you should end up with more complex looking waves and all you had to do was add a bit of math!



For more information about Spatial shaders read the Shading Language doc and the Spatial Shaders doc. Also look at more advanced tutorials in the Shading section and the 3D sections.

Shader materials

Introduction

For the most common cases, Godot provides ready to use materials for most types of shaders, such as StandardMaterial3D, CanvasItemMaterial and ParticleProcessMaterial. They are flexible implementations that cover most use cases.

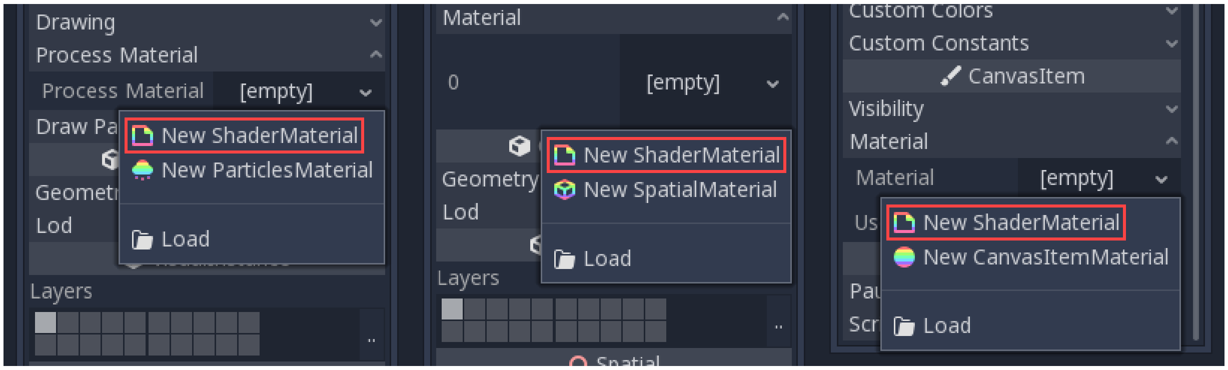
Shader materials allow writing a custom shader directly, for maximum flexibility. Examples of this are:

- Create procedural textures.
- Create complex texture blending effects.
- Create animated materials, or materials that change over time.
- Create refractive effects or other advanced effects.
- Create special lighting shaders for more exotic materials.
- Animate vertices, like tree leaves or grass.
- Create custom particle code.
- And much more!

Godot provides built in functionality to make frequent operations easier. Additionally, Godot's shader editor will detect errors as you type, so you can see your edited shaders in real-time. It is also possible to edit shaders using a visual, node-based graph editor.

Creating a ShaderMaterial

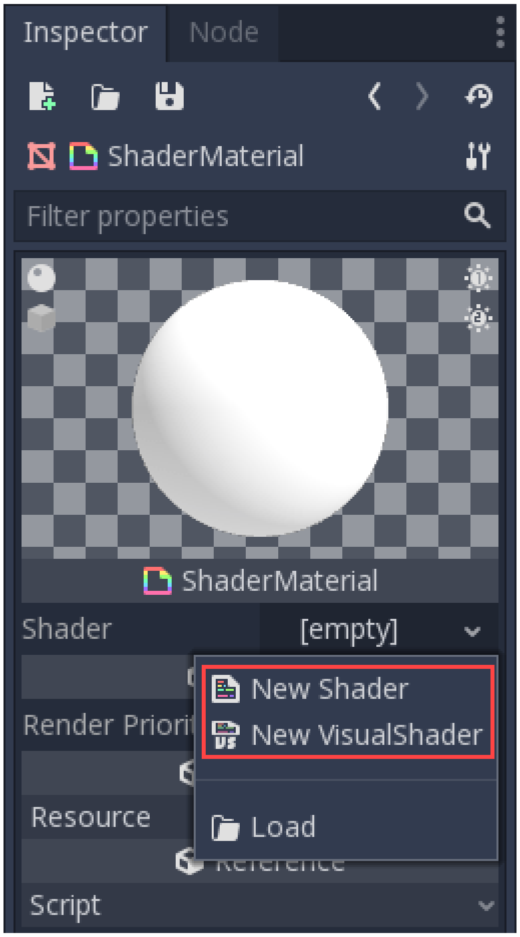
Create a new ShaderMaterial in some object of your choice. Go to the "Material" property and create a ShaderMaterial.


Drawing
Process Material
Process Material
[empty]
Draw Pa
New ShaderMaterial
New ParticlesMaterial
Geometr
Lod
Load
Layers
Material
0
[empty]
New ShaderMaterial
New SpatialMaterial
Geometry
Lod
Load
Layers
Spatial
Custom Colors
Custom Constants
CanvasItem
Visibility
Material
Material
[empty]
Us
New ShaderMaterial
New CanvasItemMaterial
Pau
Scr
Load

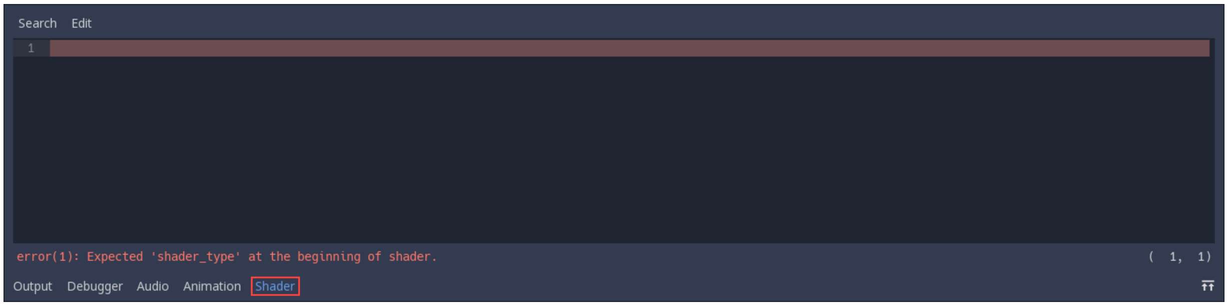
Next, click on the shader material to see its properties. From here you can create a shader or visual shader. Regular shaders use code to set their properties while a visual shader uses a node based workflow.

If you need to you can convert a visual shader to a text shader. However you can't convert a text shader to a visual shader.

Choose one of them and you can start editing your shader.

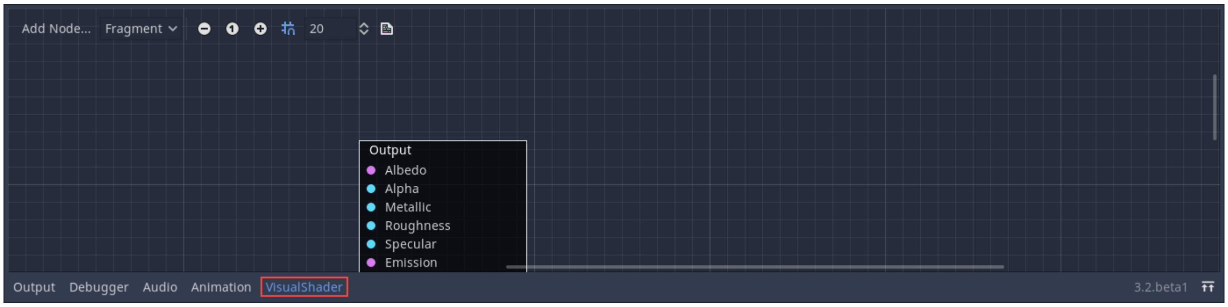

Inspector
Node
ShaderMaterial
Filter properties
ShaderMaterial
Shader
[empty]
New Shader
New VisualShader
Load
Render Priori
Resource
Reference
Script

If you create a regular shader click on it and the shader editor will open.

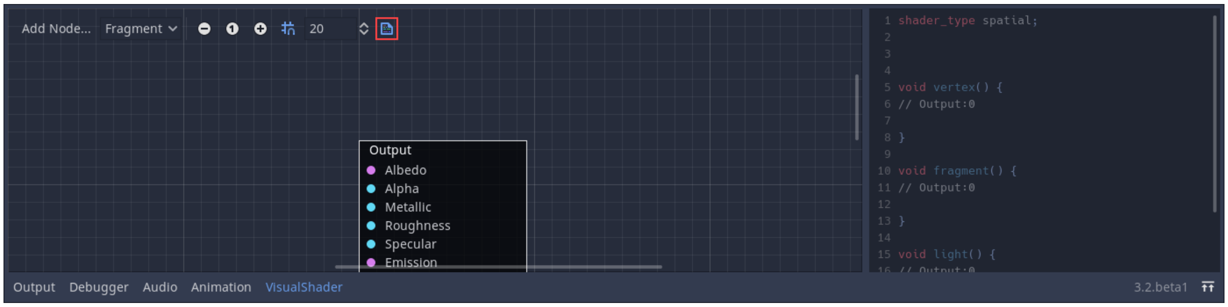

Search
Edit
Output
Debugger
Audio
Animation
Shader

```
error(1): Expected 'shader_type' at the beginning of shader.                ( 1, 1)
```

If you create a visual shader the visual shader editor will open automatically.


Add Node...
Fragment
20
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
Output
Debugger
Audio
Animation
VisualShader
3.2.beta1

In the visual shader editor you can click a button and see what your visual shader looks like as shader code. This can be useful if you're trying to replicate a text shader as nodes, or it can be used as a preview for converting your visual shader to a text shader.


Add Node...
Fragment
20
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
Output
Debugger
Audio
Animation
VisualShader
3.2.beta1

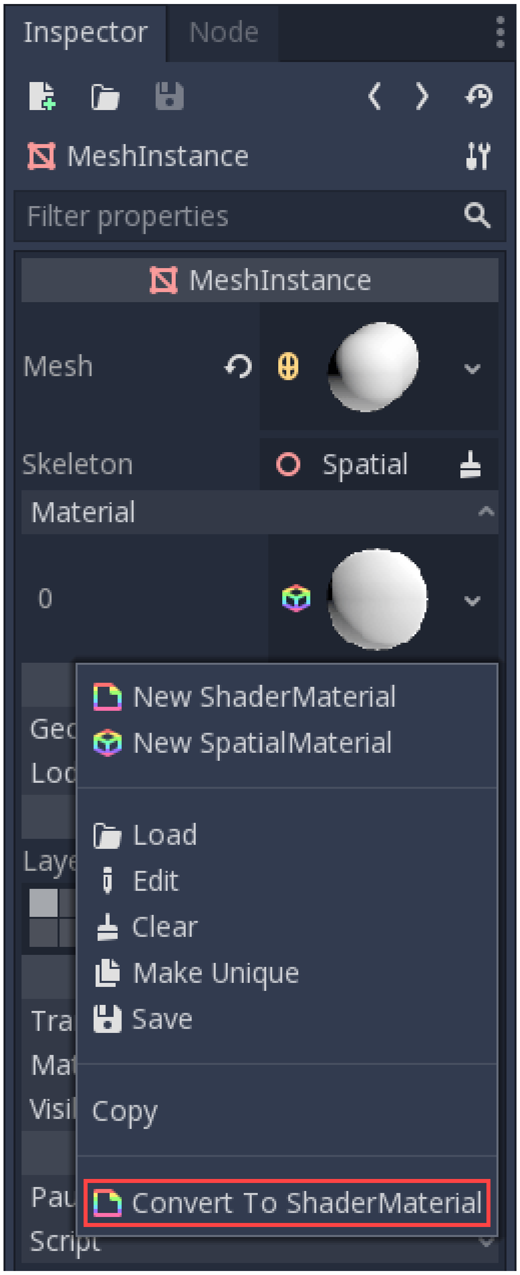
```
shader_type spatial;



void vertex() {
// Output:0

}

void fragment() {
// Output:0

}

void light() {
// Output:0
```

Note

To learn more about visual shaders, read Using VisualShaders.

Converting to ShaderMaterial

It is possible to convert from StandardMaterial3D, CanvasItemMaterial and ParticleProcessMaterial to ShaderMaterial. To do so, go to the material properties and select the convert option.



Inspector
Node
MeshInstance
Filter properties
MeshInstance
Mesh
Skeleton
Spatial
Material
0
New ShaderMaterial
New SpatialMaterial
Load
Edit
Clear
Make Unique
Save
Copy
Convert To ShaderMaterial
Script

Note

Using the convert option will turn the StandardMaterial3D into a ShaderMaterial with a text shader, not a visual shader.

Using VisualShaders

VisualShaders are the visual alternative for creating shaders.

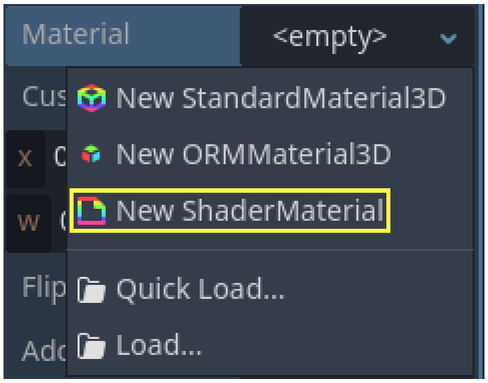
As shaders are inherently linked to visuals, the graph-based approach with previews of textures, materials, etc. offers a lot of additional convenience compared to purely script-based shaders. On the other hand, VisualShaders do not expose all features of the shader script and using both in parallel might be necessary for specific effects.

Note

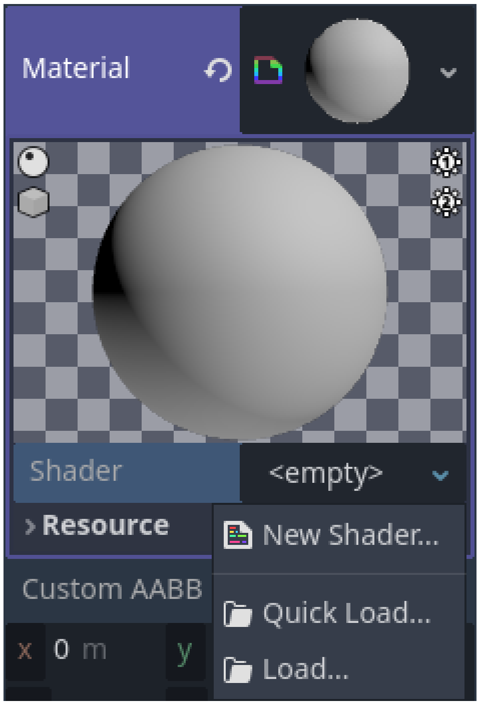
If you are not familiar with shaders, start by reading Introduction to shaders.

Creating a VisualShader

VisualShaders can be created in any ShaderMaterial. To begin using VisualShaders, create a new `ShaderMaterial` in an object of your choice.

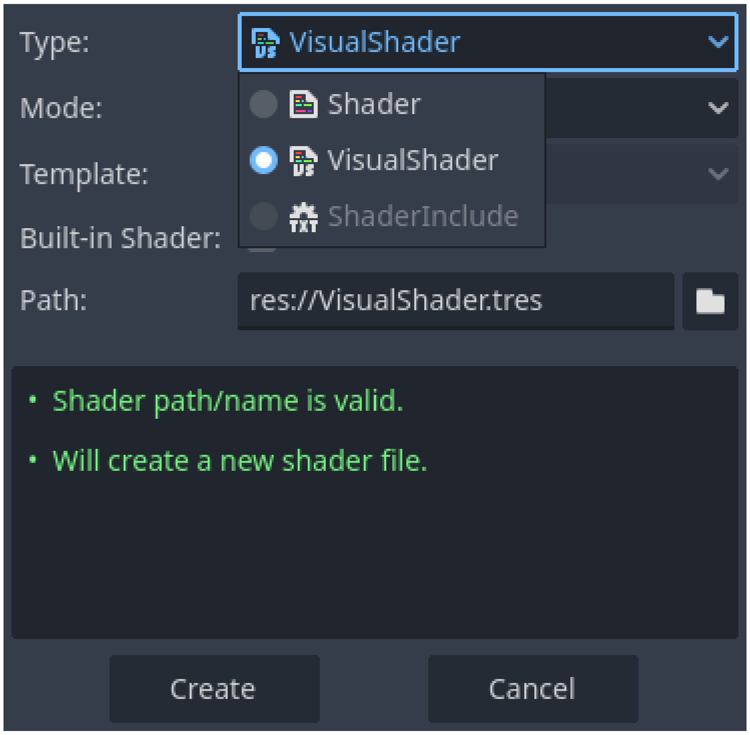

Material
<empty>
Cus
New StandardMaterial3D
x
New ORMMaterial3D
w
New ShaderMaterial
Flip
Quick Load...
Adc
Load...

Then assign a Shader resource to the `Shader` property.


Material
Resource
Custom AABB
x 0 m
y
New Shader...
Quick Load...
Load...

Shader <empty>

Click on the new `Shader` resource and the Create Shader dialog will open automatically. Change the Type option to VisualShader in the dropdown, then give it a name.

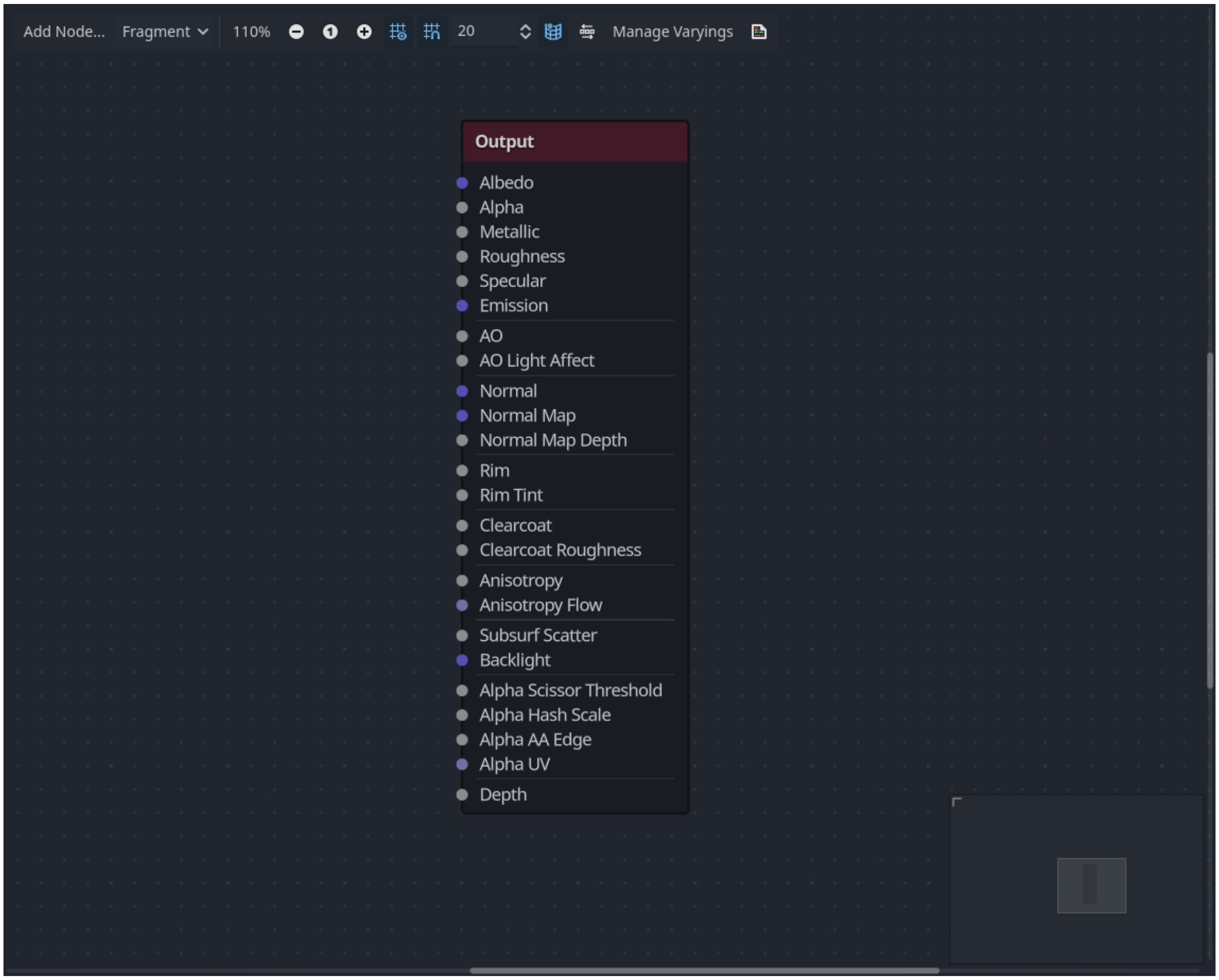

Mode:
Template:
Built-in Shader:
○ Shader
● VisualShader
○ ShaderInclude
Create
Cancel

Type: VisualShader

Path: res://VisualShader.tres

- Shader path/name is valid.
- Will create a new shader file.

Click on the visual shader you just created to open the Shader Editor. The layout of the Shader Editor comprises two parts: the upper toolbar and the graph itself.


Add Node...
Fragment
110%
20
Manage Varyings
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
AO
AO Light Affect
Normal
Normal Map
Normal Map Depth
Rim
Rim Tint
Clearcoat
Clearcoat Roughness
Anisotropy
Anisotropy Flow
Subsurf Scatter
Backlight
Alpha Scissor Threshold
Alpha Hash Scale
Alpha AA Edge
Alpha UV
Depth

From left to right in the toolbar:

- The `Add Node` button displays a popup menu to let you add nodes to the shader graph.
- The drop-down menu is the shader type: Vertex, Fragment and Light. Like for script shaders, it defines what built-in nodes will be available.
- The following buttons and number input control the zooming level, grid snapping and distance between grid lines (in pixels).

- The toggle controls if the graph minimap in the bottom right of the editor is visible or not.
- The automatically arrange selected nodes button will try to organize any nodes you have selected as efficiently and cleanly as possible.
- The Manage Varyings button opens a dropdown that lets you add or remove a varying.
- The last icon shows the generated shader code corresponding to your graph.

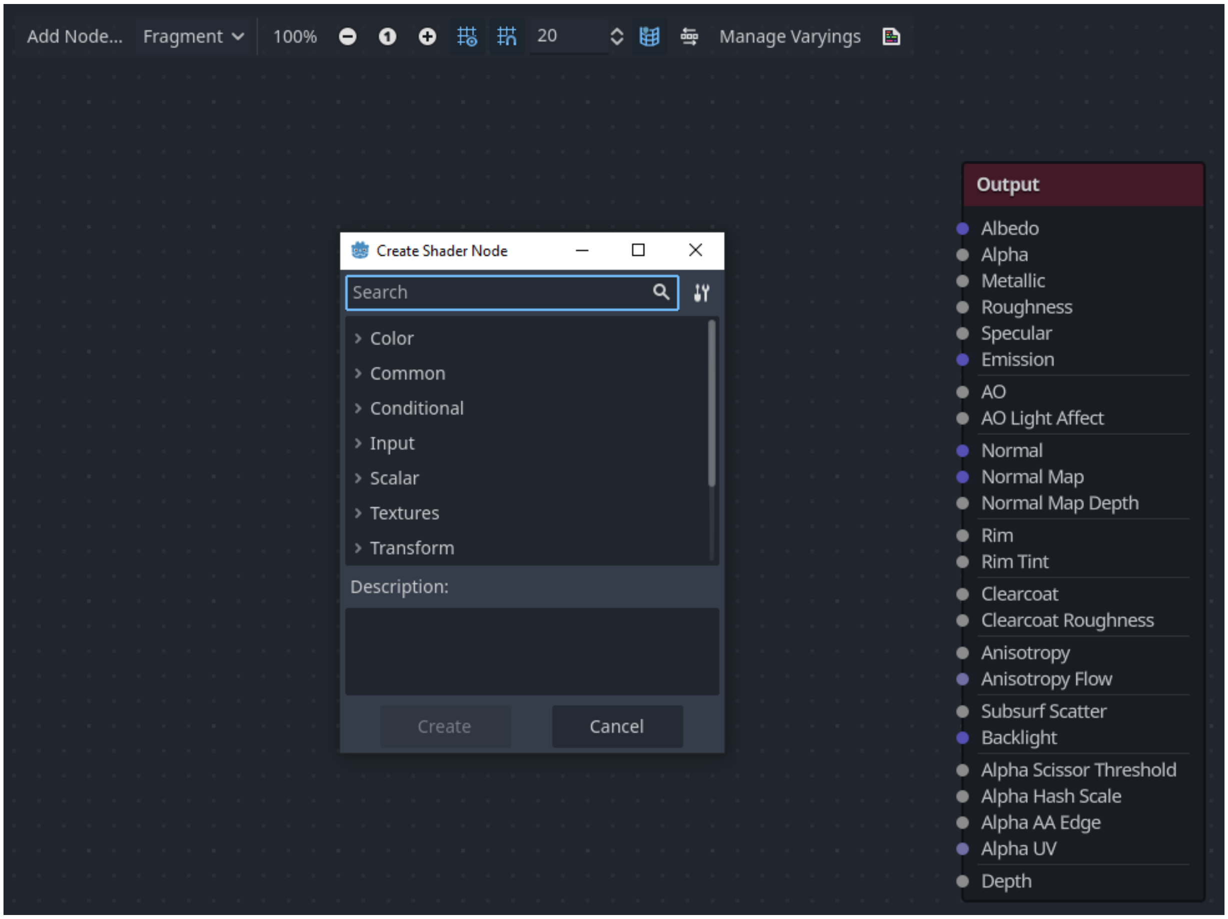
Note

Although VisualShaders do not require coding, they share the same logic with script shaders. It is advised to learn the basics of both to have a good understanding of the shading pipeline.

The visual shader graph is converted to a script shader behind the scene, and you can see this code by pressing the last button in the toolbar. This can be convenient to understand what a given node does and how to reproduce it in scripts.

Using the Visual Shader Editor

By default, every new `VisualShader` will have an output node. Every node connection ends at one of the output node's sockets. A node is the basic unit to create your shader. To add a new node, click on the `Add Node` button on the upper left corner or right click on any empty location in the graph, and a menu will pop up.


Add Node...
Fragment
100%
20
Manage Varyings
Create Shader Node
Search
Color
Common
Conditional
Input
Scalar
Textures
Transform
Description:
Create
Cancel
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
AO
AO Light Affect
Normal
Normal Map
Normal Map Depth
Rim
Rim Tint
Clearcoat
Clearcoat Roughness
Anisotropy
Anisotropy Flow
Subsurf Scatter
Backlight
Alpha Scissor Threshold
Alpha Hash Scale
Alpha AA Edge
Alpha UV
Depth

This popup has the following properties:

- If you right-click on the graph, this menu will be called at the cursor position and the created node, in that case, will also be placed under that position; otherwise, it will be created at the graph's center.
- It can be resized horizontally and vertically allowing more content to be shown. Size transform and tree content position are saved between the calls, so if you suddenly closed the popup you can easily restore its previous state.
- The `Expand All` and `Collapse All` options in the drop-down option menu can be used to easily list the available nodes.
- You can also drag and drop nodes from the popup onto the graph.

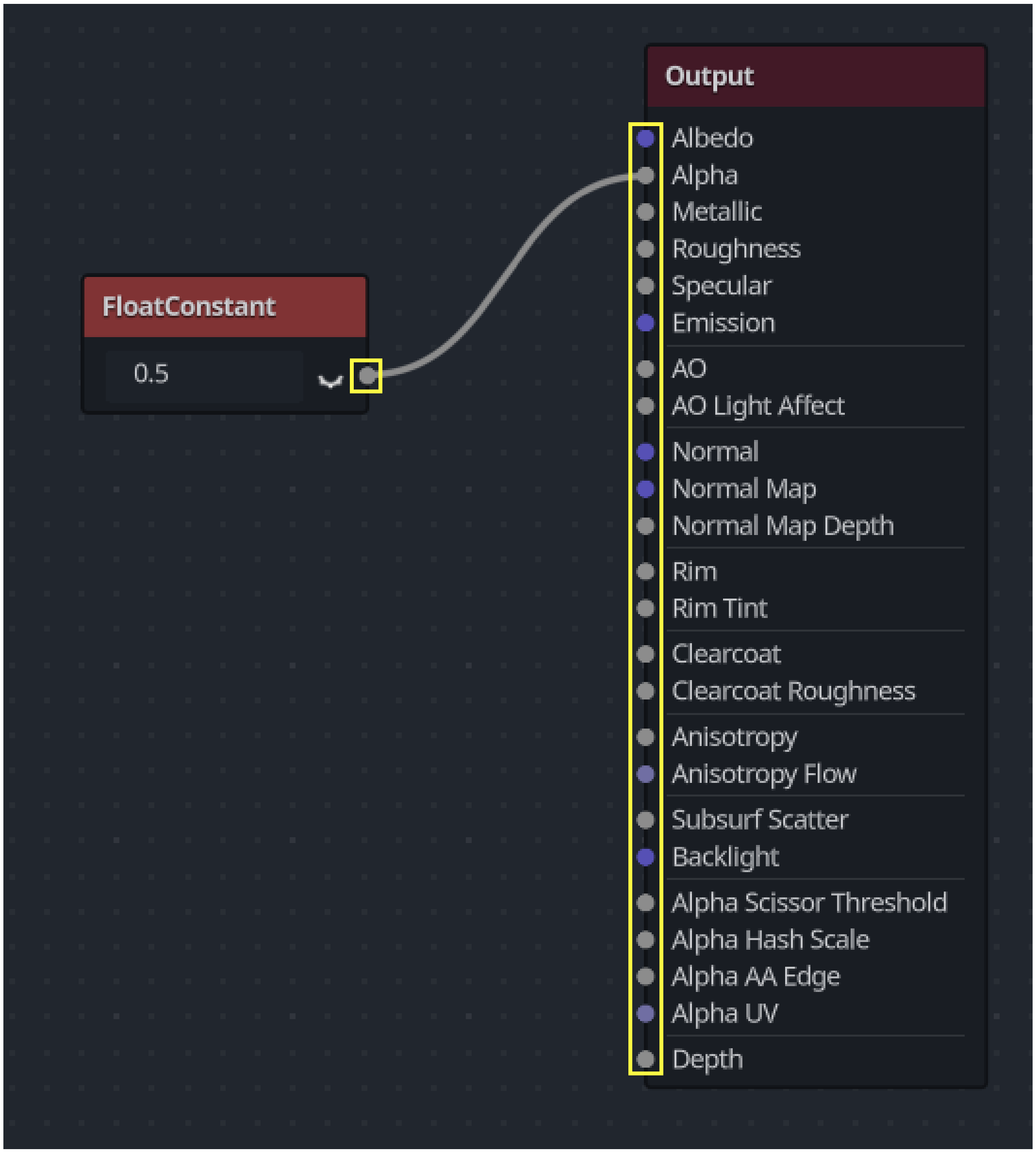
While the popup has nodes sorted in categories, it can seem overwhelming at first. Try to add some of the nodes, plug them in the output socket and observe what happens.

When connecting any `scalar` output to a `vector` input, all components of the vector will take the value of the scalar.

When connecting any `vector` output to a `scalar` input, the value of the scalar will be the average of the vector's components.

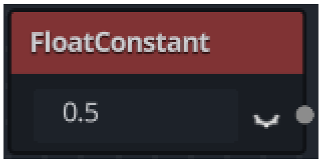
Visual Shader node interface

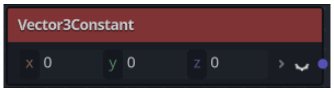
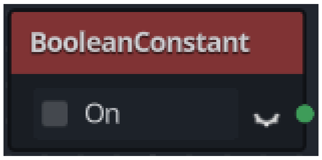
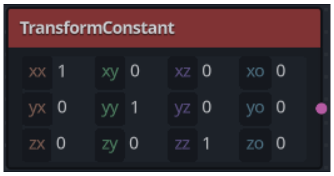
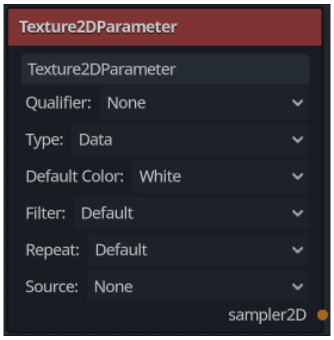
Visual shader nodes have input and output ports. The input ports are located on the left side of the node, and output ports are located on the right side of the node.


Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
AO
AO Light Affect
Normal
Normal Map
Normal Map Depth
Rim
Rim Tint
Clearcoat
Clearcoat Roughness
Anisotropy
Anisotropy Flow
Subsurf Scatter
Backlight
Alpha Scissor Threshold
Alpha Hash Scale
Alpha AA Edge
Alpha UV
Depth
FloatConstant
0.5

These ports are colored to differentiate type of port:

Port types

Type	**Color**	**Description**	**Example**
Scalar	Gray	Scalar is a single value.	 FloatConstant 0.5

Type	Color	Description	Example
Vector	Purple	Vector is a set of values.	 Vector3Constant x 0 y 0 z 0
Boolean	Green	On or off, true or false.	 BooleanConstant ☐ On
Transform	Pink	A matrix, usually used to transform vertices.	 TransformConstant xx 1 xy 0 xz 0 xo 0 yx 0 yy 1 yz 0 yo 0 zx 0 zy 0 zz 1 zo 0
Sampler	Orange	A texture sampler. It can be used to sample textures.	 Texture2DParameter Texture2DParameter Qualifier: None Type: Data Default Color: White Filter: Default Repeat: Default Source: None sampler2D

All of the types are used in the calculations of vertices, fragments, and lights in the shader. For example: matrix multiplication, vector addition, or scalar division.

There are other types but these are the main ones.

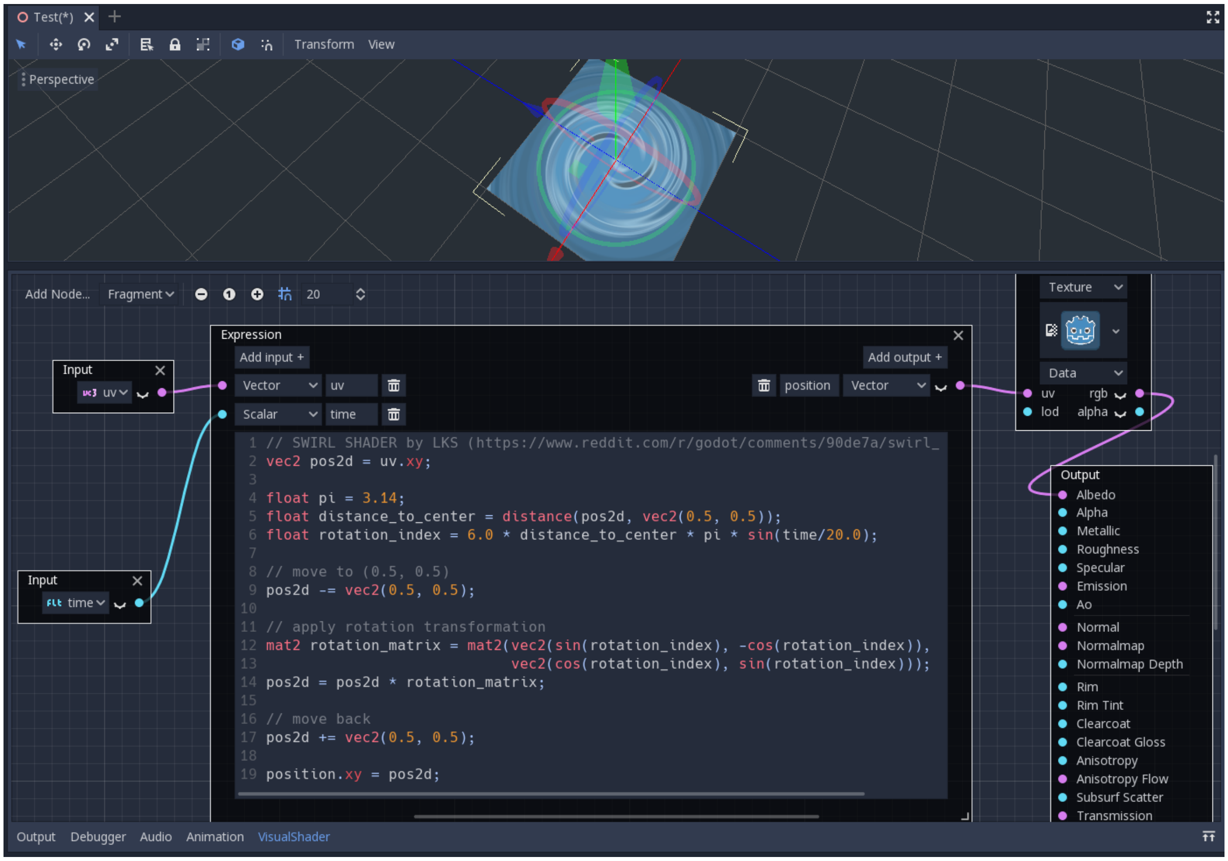
Visual Shader nodes

Below are some special nodes that are worth knowing about. The list is not exhaustive and might be expanded with more nodes and examples.

Expression node

The `Expression` node allows you to write Godot Shading Language (GLSL-like) expressions inside your visual shaders. The node has buttons to add any amount of required input and output ports and can be resized. You can also set up the name and type of each port. The expression you have entered will apply immediately to the material (once the focus leaves the expression text box). Any parsing or compilation errors will be printed to the Output tab. The outputs are initialized to their zero value by default. The node is located under the Special tab and can be used in all shader modes.

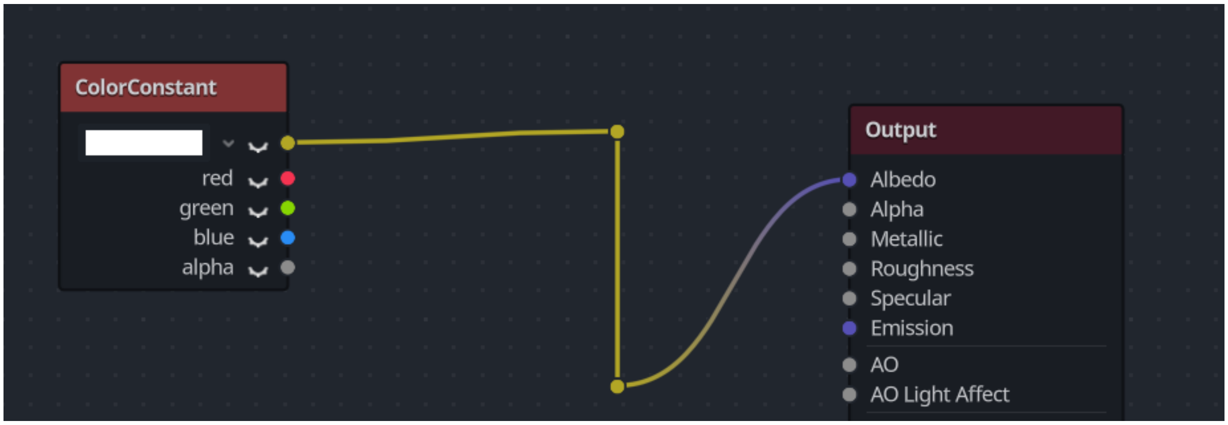
The possibilities of this node are almost limitless – you can write complex procedures, and use all the power of text-based shaders, such as loops, the `discard` keyword, extended types, etc. For example:


Test(*)
Transform
View
Perspective
Add Node...
Fragment
20
Input
uv
Input
time
Expression
Add input +
Vector
uv
Scalar
time
Add output +
position
Vector
Texture
Data
uv
rgb
lod
alpha
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
Ao
Normal
Normalmap
Normalmap Depth
Rim
Rim Tint
Clearcoat
Clearcoat Gloss
Anisotropy
Anisotropy Flow
Subsurf Scatter
Transmission
Output
Debugger
Audio
Animation
VisualShader

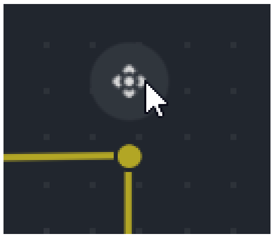
```
// SWIRL SHADER by LKS (https://www.reddit.com/r/godot/comments/90de7a/swirl_
vec2 pos2d = uv.xy;

float pi = 3.14;
float distance_to_center = distance(pos2d, vec2(0.5, 0.5));
float rotation_index = 6.0 * distance_to_center * pi * sin(time/20.0);

// move to (0.5, 0.5)
pos2d -= vec2(0.5, 0.5);

// apply rotation transformation
mat2 rotation_matrix = mat2(vec2(sin(rotation_index), -cos(rotation_index)),
                            vec2(cos(rotation_index), sin(rotation_index)));
pos2d = pos2d * rotation_matrix;

// move back
pos2d += vec2(0.5, 0.5);

position.xy = pos2d;
```

Reroute node

The `Reroute` node is used purely for organizational purposes. In a complicated shader with many nodes you may find that the paths between nodes can make things hard to read. Reroute, as its name suggests, allows you to adjust the path between nodes to make things easier to read. You can even have multiple reroute nodes for a single path, which can be used to make right angles.

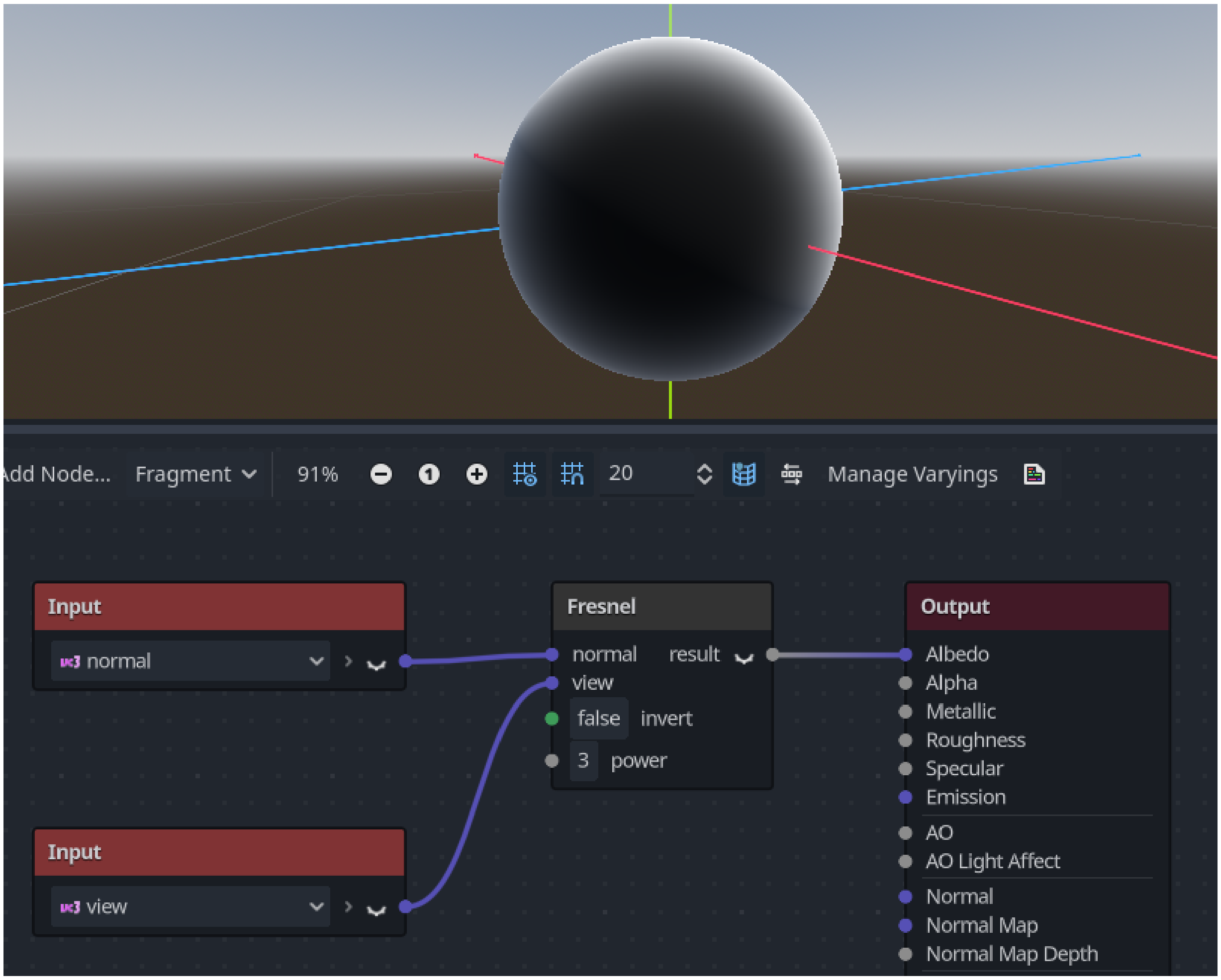

ColorConstant
red
green
blue
alpha
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
AO
AO Light Affect

To move a reroute node move your mouse cursor above it, and grab the handle that appears.



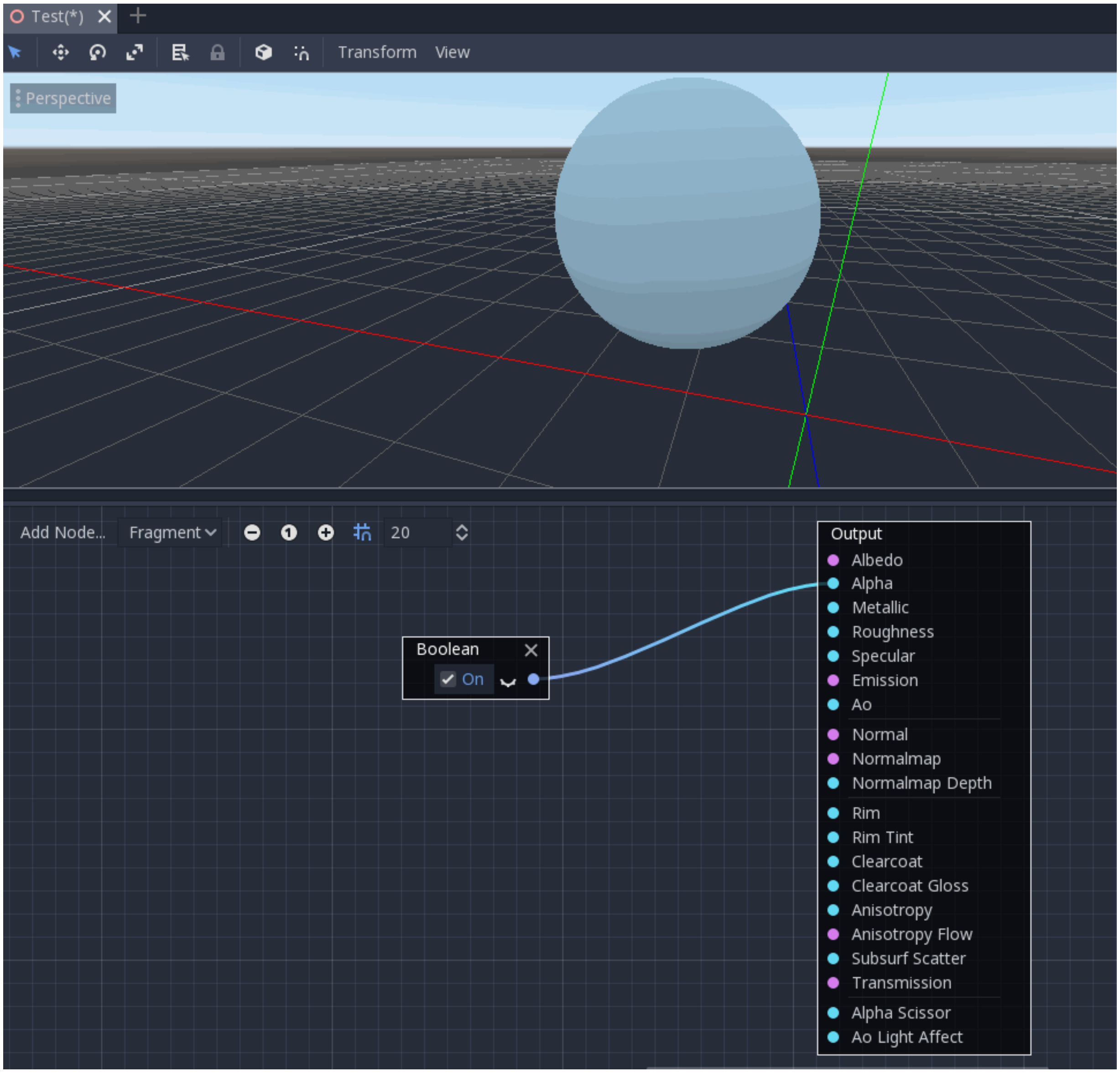
Fresnel node

The `Fresnel` node is designed to accept normal and view vectors and produces a scalar which is the saturated dot product between them. Additionally, you can setup the inversion and the power of equation. The `Fresnel` node is great for adding a rim-like lighting effect to objects.


Add Node...
Fragment
91%
20
Manage Varyings
Input
normal
Fresnel
normal
result
view
false
invert
3
power
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
AO
AO Light Affect
Normal
Normal Map
Normal Map Depth
Input
view

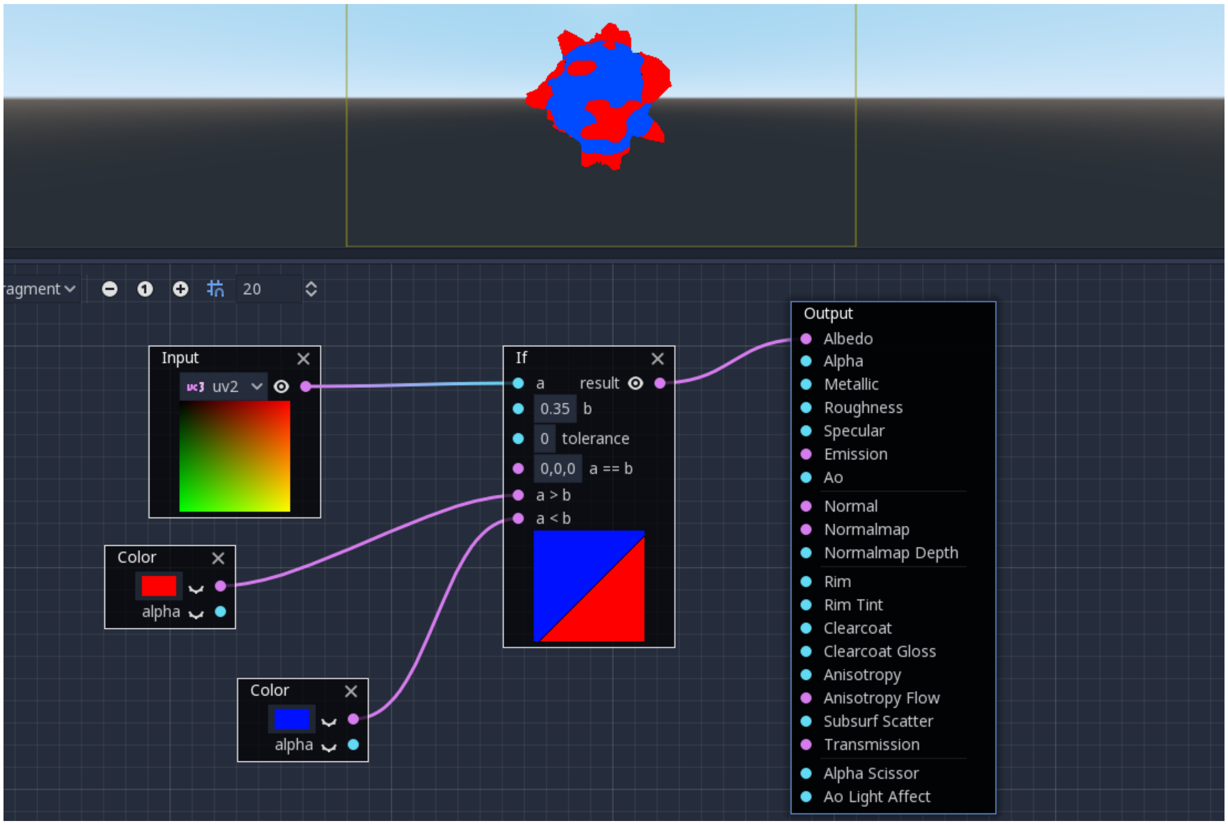
Boolean node

The `Boolean` node can be converted to `Scalar` or `Vector` to represent `0` or `1` and `(0, 0, 0)` or `(1, 1, 1)` respectively. This property can be used to enable or disable some effect parts with one click.


Test(*)
Transform
View
Perspective
Add Node...
Fragment
20
Boolean
On
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
Ao
Normal
Normalmap
Normalmap Depth
Rim
Rim Tint
Clearcoat
Clearcoat Gloss
Anisotropy
Anisotropy Flow
Subsurf Scatter
Transmission
Alpha Scissor
Ao Light Affect

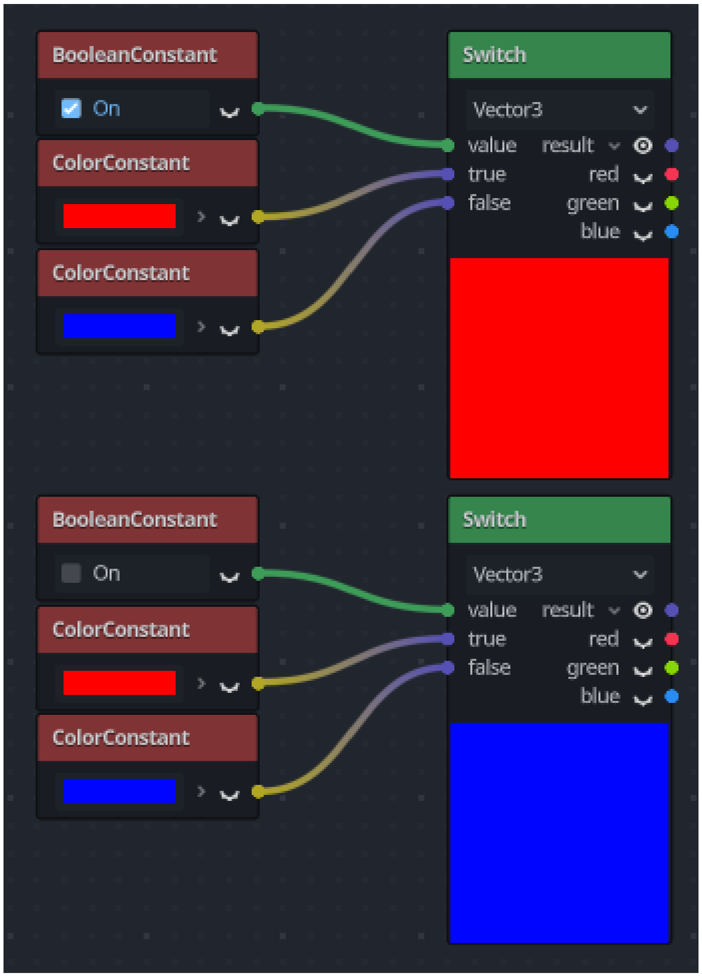
If node

The `If` node allows you to setup a vector which will be returned the result of the comparison between `a` and `b`. There are three vectors which can be returned: `a == b` (in that case the tolerance parameter is provided as a comparison threshold – by default it is equal to the minimal value, i.e. `0.00001`), `a > b` and `a < b`.


agment
20
Input
uv2
Color
alpha
Color
alpha
If
a
result
0.35
b
0
tolerance
0,0,0
a == b
a > b
a < b
Output
Albedo
Alpha
Metallic
Roughness
Specular
Emission
Ao
Normal
Normalmap
Normalmap Depth
Rim
Rim Tint
Clearcoat
Clearcoat Gloss
Anisotropy
Anisotropy Flow
Subsurf Scatter
Transmission
Alpha Scissor
Ao Light Affect

Switch node

The `Switch` node returns a vector if the boolean condition is `true` or `false`. `Boolean` was introduced above. If you convert a vector to a true boolean, all components of the vector should be above zero.


BooleanConstant
☑ On
ColorConstant
ColorConstant
Switch
Vector3
value
result
true
red
false
green
blue
BooleanConstant
☐ On
ColorConstant
ColorConstant
Switch
Vector3
value
result
true
red
false
green
blue

Using compute shaders

This tutorial will walk you through the process of creating a minimal compute shader. But first, a bit of background on compute shaders and how they work with Godot.

Note

This tutorial assumes you are familiar with shaders generally. If you are new to shaders please read Introduction to shaders and your first shader before proceeding with this tutorial.

A compute shader is a special type of shader program that is orientated towards general purpose programming. In other words, they are more flexible than vertex shaders and fragment shaders as they don't have a fixed purpose (i.e. transforming vertices or writing colors to an image). Unlike fragment shaders and vertex shaders, compute shaders have very little going on behind the scenes. The code you write is what the GPU runs and very little else. This can make them a very useful tool to offload heavy calculations to the GPU.

Now let's get started by creating a short compute shader.

First, in the **external** text editor of your choice, create a new file called `compute_example.glsl` in your project folder. When you write compute shaders in Godot, you write them in GLSL directly. The Godot shader language is based on GLSL. If you are familiar with normal shaders in Godot, the syntax below will look somewhat familiar.

Note

Compute shaders can only be used from RenderingDevice-based renderers (the Forward+ or Mobile renderer). To follow along with this tutorial, ensure that you are using the Forward+ or Mobile renderer. The setting for which is located in the top right-hand corner of the editor.

Note that compute shader support is generally poor on mobile devices (due to driver bugs), even if they are technically supported.

Let's take a look at this compute shader code:

```
#[compute]
#version 450

// Invocations in the (x, y, z) dimension
layout(local_size_x = 2, local_size_y = 1, local_size_z = 1)
in;

// A binding to the buffer we create in our script
layout(set = 0, binding = 0, std430) restrict buffer
MyDataBuffer {
    float data[];
}
my_data_buffer;

// The code we want to execute in each invocation
void main() {
    // gl_GlobalInvocationID.x uniquely identifies this
invocation across all work groups
    my_data_buffer.data[gl_GlobalInvocationID.x] *= 2.0;
}
```

This code takes an array of floats, multiplies each element by 2 and store the results back in the buffer array. Now let's look at it line-by-line.

```
#[compute]
```

```
#version 450
```

These two lines communicate two things:

1. The following code is a compute shader. This is a Godot-specific hint that is needed for the editor to properly import the shader file.
2. The code is using GLSL version 450.

You should never have to change these two lines for your custom compute shaders.

```
// Invocations in the (x, y, z) dimension
layout(local_size_x = 2, local_size_y = 1, local_size_z = 1)
in;
```

Next, we communicate the number of invocations to be used in each workgroup. Invocations are instances of the shader that are running within the same workgroup. When we launch a compute shader from the CPU, we tell it how many workgroups to run. Workgroups run in parallel to each other. While running one workgroup, you cannot access information in another workgroup. However, invocations in the same workgroup can have some limited access to other invocations.

Think about workgroups and invocations as a giant nested `for` loop.

```
for (int x = 0; x < workgroup_size_x; x++) {
  for (int y = 0; y < workgroup_size_y; y++) {
     for (int z = 0; z < workgroup_size_z; z++) {
        // Each workgroup runs independently and in parallel.
        for (int local_x = 0; local_x < invocation_size_x;
local_x++) {
           for (int local_y = 0; local_y < invocation_size_y;
local_y++) {
              for (int local_z = 0; local_z <
invocation_size_z; local_z++) {
                 // Compute shader runs here.
              }
```

```
            }
         }
      }
   }
}
```

Workgroups and invocations are an advanced topic. For now, remember that we will be running two invocations per workgroup.

```
// A binding to the buffer we create in our script
layout(set = 0, binding = 0, std430) restrict buffer
MyDataBuffer {
    float data[];
}
my_data_buffer;
```

Here we provide information about the memory that the compute shader will have access to. The `layout` property allows us to tell the shader where to look for the buffer, we will need to match these `set` and `binding` positions from the CPU side later.

The `restrict` keyword tells the shader that this buffer is only going to be accessed from one place in this shader. In other words, we won't bind this buffer in another `set` or `binding` index. This is important as it allows the shader compiler to optimize the shader code. Always use `restrict` when you can.

This is an *unsized* buffer, which means it can be any size. So we need to be careful not to read from an index larger than the size of the buffer.

```
// The code we want to execute in each invocation
void main() {
    // gl_GlobalInvocationID.x uniquely identifies this
invocation across all work groups
    my_data_buffer.data[gl_GlobalInvocationID.x] *= 2.0;
}
```

Finally, we write the `main` function which is where all the logic happens. We access a position in the storage buffer using the `gl_GlobalInvocationID` built in variables. `gl_GlobalInvocationID` gives you the global unique ID for the current invocation.

To continue, write the code above into your newly created `compute_example.glsl` file.

Create a local RenderingDevice

To interact with and execute a compute shader, we need a script. Create a new script in the language of your choice and attach it to any Node in your scene.

Now to execute our shader we need a local RenderingDevice which can be created using the RenderingServer:

GDScript

```
# Create a local rendering device.
var rd := RenderingServer.create_local_rendering_device()
```

C#

```
// Create a local rendering device.
var rd = RenderingServer.CreateLocalRenderingDevice();
```

After that, we can load the newly created shader file `compute_example.glsl` and create a precompiled version of it using this:

GDScript

```
# Load GLSL shader
var shader_file := load("res://compute_example.glsl")
var shader_spirv: RDShaderSPIRV = shader_file.get_spirv()
var shader := rd.shader_create_from_spirv(shader_spirv)
```

C#

```
// Load GLSL shader
var shaderFile = GD.Load<RDShaderFile>
("res://compute_example.glsl");
var shaderBytecode = shaderFile.GetSpirV();
var shader = rd.ShaderCreateFromSpirV(shaderBytecode);
```

Warning

Local RenderingDevices cannot be debugged using tools such as RenderDoc [https://renderdoc.org/].

Provide input data

As you might remember, we want to pass an input array to our shader, multiply each element by 2 and get the results.

We need to create a buffer to pass values to a compute shader. We are dealing with an array of floats, so we will use a storage buffer for this example. A storage buffer takes an array of bytes and allows the CPU to transfer data to and from the GPU.

So let's initialize an array of floats and create a storage buffer:

GDScript

```
# Prepare our data. We use floats in the shader, so we need 32
bit.
var input := PackedFloat32Array([1, 2, 3, 4, 5, 6, 7, 8, 9,
10])
var input_bytes := input.to_byte_array()
```

```
# Create a storage buffer that can hold our float values.
# Each float has 4 bytes (32 bit) so 10 x 4 = 40 bytes
var buffer := rd.storage_buffer_create(input_bytes.size(),
input_bytes)
```

C#

```
// Prepare our data. We use floats in the shader, so we need 32
bit.
var input = new float[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var inputBytes = new byte[input.Length * sizeof(float)];
Buffer.BlockCopy(input, 0, inputBytes, 0, inputBytes.Length);

// Create a storage buffer that can hold our float values.
// Each float has 4 bytes (32 bit) so 10 x 4 = 40 bytes
var buffer = rd.StorageBufferCreate((uint)inputBytes.Length,
inputBytes);
```

With the buffer in place we need to tell the rendering device to use this buffer. To do that we will need to create a uniform (like in normal shaders) and assign it to a uniform set which we can pass to our shader later.

GDScript

```
# Create a uniform to assign the buffer to the rendering device
var uniform := RDUniform.new()
uniform.uniform_type =
RenderingDevice.UNIFORM_TYPE_STORAGE_BUFFER
uniform.binding = 0 # this needs to match the "binding" in our
shader file
uniform.add_id(buffer)
var uniform_set := rd.uniform_set_create([uniform], shader, 0)
# the last parameter (the 0) needs to match the "set" in our
shader file
```

C#

```
// Create a uniform to assign the buffer to the rendering
device
```

```
var uniform = new RDUniform
{
    UniformType = RenderingDevice.UniformType.StorageBuffer,
    Binding = 0
};
uniform.AddId(buffer);
var uniformSet = rd.UniformSetCreate(new Array<RDUniform> {
uniform }, shader, 0);
```

Defining a compute pipeline

The next step is to create a set of instructions our GPU can execute. We need a pipeline and a compute list for that.

The steps we need to do to compute our result are:

1. Create a new pipeline.
2. Begin a list of instructions for our GPU to execute.
3. Bind our compute list to our pipeline
4. Bind our buffer uniform to our pipeline
5. Specify how many workgroups to use
6. End the list of instructions

GDScript

```
# Create a compute pipeline
var pipeline := rd.compute_pipeline_create(shader)
var compute_list := rd.compute_list_begin()
rd.compute_list_bind_compute_pipeline(compute_list, pipeline)
rd.compute_list_bind_uniform_set(compute_list, uniform_set, 0)
rd.compute_list_dispatch(compute_list, 5, 1, 1)
rd.compute_list_end()
```

C#

```
// Create a compute pipeline
var pipeline = rd.ComputePipelineCreate(shader);
var computeList = rd.ComputeListBegin();
```

```
rd.ComputeListBindComputePipeline(computeList, pipeline);
rd.ComputeListBindUniformSet(computeList, uniformSet, 0);
rd.ComputeListDispatch(computeList, xGroups: 5, yGroups: 1,
zGroups: 1);
rd.ComputeListEnd();
```

Note that we are dispatching the compute shader with 5 work groups in the X axis, and one in the others. Since we have 2 local invocations in the X axis (specified in our shader), 10 compute shader invocations will be launched in total. If you read or write to indices outside of the range of your buffer, you may access memory outside of your shaders control or parts of other variables which may cause issues on some hardware.

Execute a compute shader

After all of this we are almost done, but we still need to execute our pipeline. So far we have only recorded what we would like the GPU to do; we have not actually run the shader program.

To execute our compute shader we need to submit the pipeline to the GPU and wait for the execution to finish:

GDScript

```
# Submit to GPU and wait for sync
rd.submit()
rd.sync()
```

C#

```
// Submit to GPU and wait for sync
rd.Submit();
rd.Sync();
```

Ideally, you would not call `sync()` to synchronize the RenderingDevice right away as it will cause the CPU to wait for the

GPU to finish working. In our example, we synchronize right away because we want our data available for reading right away. In general, you will want to wait *at least* 2 or 3 frames before synchronizing so that the GPU is able to run in parallel with the CPU.

Warning

Long computations can cause Windows graphics drivers to "crash" due to TDR being triggered by Windows. This is a mechanism that reinitializes the graphics driver after a certain amount of time has passed without any activity from the graphics driver (usually 5 to 10 seconds).

Depending on the duration your compute shader takes to execute, you may need to split it into multiple dispatches to reduce the time each dispatch takes and reduce the chances of triggering a TDR. Given TDR is time-dependent, slower GPUs may be more prone to TDRs when running a given compute shader compared to a faster GPU.

Retrieving results

You may have noticed that, in the example shader, we modified the contents of the storage buffer. In other words, the shader read from our array and stored the data in the same array again so our results are already there. Let's retrieve the data and print the results to our console.

GDScript

```
# Read back the data from the buffer
var output_bytes := rd.buffer_get_data(buffer)
var output := output_bytes.to_float32_array()
```

```
print("Input: ", input)
print("Output: ", output)
```

C#

```
// Read back the data from the buffers
var outputBytes = rd.BufferGetData(buffer);
var output = new float[input.Length];
Buffer.BlockCopy(outputBytes, 0, output, 0,
outputBytes.Length);
GD.Print("Input: ", string.Join(", ", input));
GD.Print("Output: ", string.Join(", ", output));
```

With that, you have everything you need to get started working with compute shaders.

See also

The demo projects repository contains a Compute Shader Heightmap demo [https://github.com/godotengine/godot-demo-projects/tree/master/misc/compute_shader_heightmap] This project performs heightmap image generation on the CPU and GPU separately, which lets you compare how a similar algorithm can be implemented in two different ways (with the GPU implementation being faster in most cases).

Screen-reading shaders

Introduction

It is often desired to make a shader that reads from the same screen to which it's writing. 3D APIs, such as OpenGL or DirectX, make this very difficult because of internal hardware limitations. GPUs are extremely parallel, so reading and writing causes all sorts of cache and coherency problems. As a result, not even the most modern hardware supports this properly.

The workaround is to make a copy of the screen, or a part of the screen, to a back-buffer and then read from it while drawing. Godot provides a few tools that make this process easy.

Screen texture

Godot Shading language has a special texture to access the already rendered contents of the screen. It is used by specifying a hint when declaring a `sampler2D` uniform: `hint_screen_texture`. A special built-in varying `SCREEN_UV` can be used to obtain the UV relative to the screen for the current fragment. As a result, this canvas_item fragment shader results in an invisible object, because it only shows what lies behind:

```
shader_type canvas_item;

uniform sampler2D screen_texture : hint_screen_texture,
repeat_disable, filter_nearest;

void fragment() {
    COLOR = textureLod(screen_texture, SCREEN_UV, 0.0);
}
```

`textureLod` is used here as we only want to read from the bottom mipmap. If you want to read from a blurred version of the texture instead, you can increase the third argument to `textureLod` and change the hint `filter_nearest` to `filter_nearest_mipmap` (or any other filter with mipmaps enabled). If using a filter with mipmaps, Godot will automatically calculate the blurred texture for you.

Warning

If the filter mode is not changed to a filter mode that contains `mipmap` in its name, `textureLod` with an LOD parameter greater than `0.0` will have the same appearance as with the `0.0` LOD parameter.

Screen texture example

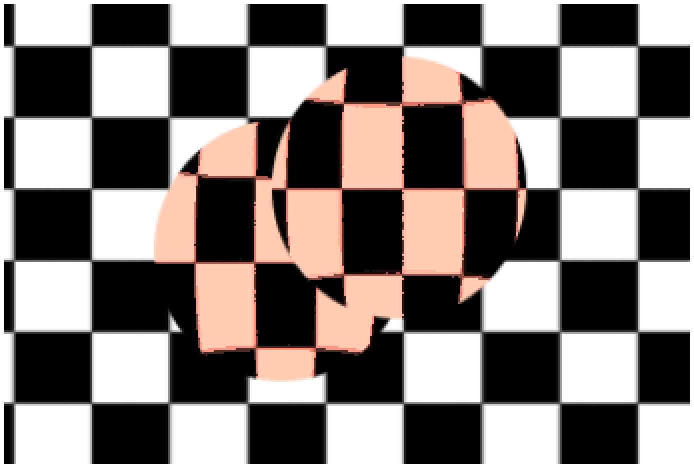
The screen texture can be used for many things. There is a special demo for *Screen Space Shaders*, that you can download to see and learn. One example is a simple shader to adjust brightness, contrast and saturation:

```
shader_type canvas_item;

uniform sampler2D screen_texture : hint_screen_texture,
repeat_disable, filter_nearest;

uniform float brightness = 1.0;
uniform float contrast = 1.0;
uniform float saturation = 1.0;

void fragment() {
    vec3 c = textureLod(screen_texture, SCREEN_UV, 0.0).rgb;
    c.rgb = mix(vec3(0.0), c.rgb, brightness);
    c.rgb = mix(vec3(0.5), c.rgb, contrast);
    c.rgb = mix(vec3(dot(vec3(1.0), c.rgb) * 0.33333), c.rgb,
saturation);

    COLOR.rgb = c;
}
```

Behind the scenes

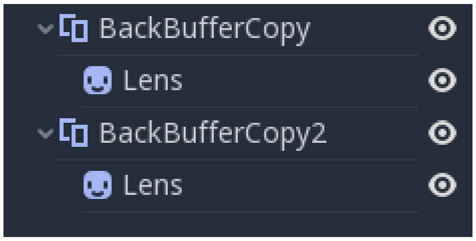
While this seems magical, it's not. In 2D, when `hint_screen_texture` is first found in a node that is about to be drawn, Godot does a full-screen copy to a back-buffer. Subsequent nodes that use it in shaders will not have the screen copied for them, because this ends up being inefficient. In 3D, the screen is copied after the opaque geometry pass, but before the transparent geometry pass, so transparent objects will not be captured in the screen texture.

As a result, in 2D, if shaders that use `hint_screen_texture` overlap, the second one will not use the result of the first one, resulting in unexpected visuals:

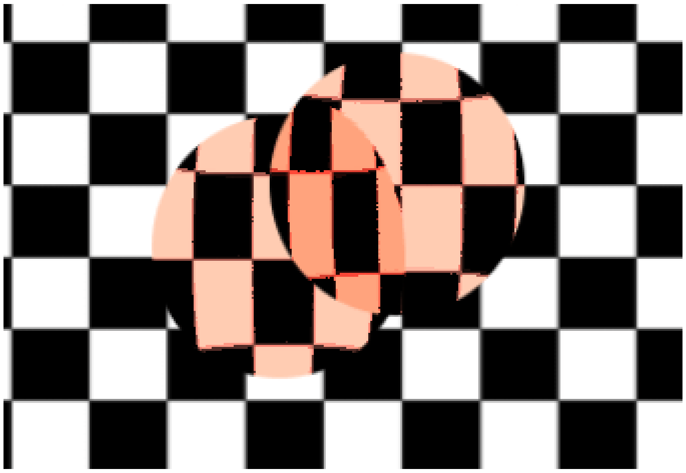


In the above image, the second sphere (top right) is using the same source for the screen texture as the first one below, so the first one "disappears", or is not visible.

In 2D, this can be corrected via the BackBufferCopy node, which can be instantiated between both spheres. BackBufferCopy can work by either specifying a screen region or the whole screen:



With correct back-buffer copying, the two spheres blend correctly:



Warning

In 3D, materials that use `hint_screen_texture` are considered transparent themselves and will not appear in the resulting screen texture of other materials. If you plan to instance a scene that uses a material with `hint_screen_texture`, you will need to use a BackBufferCopy node.

In 3D, there is less flexibility to solve this particular issue because the screen texture is only captured once. Be careful when using the screen texture in 3D as it won't capture transparent objects and may capture some opaque objects that are in front of the object using the screen texture.

You can reproduce the back-buffer logic in 3D by creating a Viewport with a camera in the same position as your object, and then use the Viewport's texture instead of the screen texture.

Back-buffer logic

So, to make it clearer, here's how the backbuffer copying logic works in 2D in Godot:

- If a node uses `hint_screen_texture`, the entire screen is copied to the back buffer before drawing that node. This only happens the first time; subsequent nodes do not trigger this.
- If a BackBufferCopy node was processed before the situation in the point above (even if `hint_screen_texture` was not used), the behavior described in the point above does not happen. In other words, automatic copying of the entire screen only happens if `hint_screen_texture` is used in a node for the first time and no BackBufferCopy node (not disabled) was found before in tree-order.
- BackBufferCopy can copy either the entire screen or a region. If set to only a region (not the whole screen) and your shader uses pixels not in the region copied, the result of that read is undefined (most likely garbage from previous frames). In other words, it's possible to use BackBufferCopy to copy back a region of the screen and then read the screen texture from a different region. Avoid this behavior!

Depth texture

For 3D shaders, it's also possible to access the screen depth buffer. For this, the `hint_depth_texture` hint is used. This texture is not linear; it must be converted using the inverse projection matrix.

The following code retrieves the 3D position below the pixel being drawn:

```
uniform sampler2D depth_texture : hint_depth_texture,
repeat_disable, filter_nearest;

void fragment() {
    float depth = textureLod(depth_texture, SCREEN_UV, 0.0).r;
    vec4 upos = INV_PROJECTION_MATRIX * vec4(SCREEN_UV * 2.0 -
1.0, depth, 1.0);
    vec3 pixel_position = upos.xyz / upos.w;
}
```

Normal-roughness texture

Note

Normal-roughness texture is only supported in the Forward+ rendering method, not Mobile or Compatibility.

Similarly, the normal-roughness texture can be used to read the normals and roughness of objects rendered in the depth prepass. The normal is stored in the `.xyz` channels (mapped to the 0-1 range) while the roughness is stored in the `.w` channel.

```
uniform sampler2D normal_roughness_texture :
hint_normal_roughness_texture, repeat_disable, filter_nearest;

void fragment() {
    float screen_roughness = texture(normal_roughness_texture,
SCREEN_UV).w;
    vec3 screen_normal = texture(normal_roughness_texture,
SCREEN_UV).xyz;
    screen_normal = screen_normal * 2.0 - 1.0;
```

Redefining screen textures

The screen texture hints (`hint_screen_texture`, `hint_depth_texture`, and `hint_normal_roughness_texture`) can be used with multiple uniforms. For example, you may want to read from the texture multiple times with a different repeat flag or filter flag.

The following example shows a shader that reads the screen space normal with linear filtering, but reads the screen space roughness using nearest neighbor filtering.

```
uniform sampler2D normal_roughness_texture :
hint_normal_roughness_texture, repeat_disable, filter_nearest;
uniform sampler2D normal_roughness_texture2 :
hint_normal_roughness_texture, repeat_enable, filter_linear;

void fragment() {
    float screen_roughness = texture(normal_roughness_texture,
SCREEN_UV).w;
    vec3 screen_normal = texture(normal_roughness_texture2,
SCREEN_UV).xyz;
    screen_normal = screen_normal * 2.0 - 1.0;
```

Converting GLSL to Godot shaders

This document explains the differences between Godot's shading language and GLSL and gives practical advice on how to migrate shaders from other sources, such as Shadertoy and The Book of Shaders, into Godot shaders.

For detailed information on Godot's shading language, please refer to the Shading Language reference.

GLSL

Godot uses a shading language based on GLSL with the addition of a few quality-of-life features. Accordingly, most features available in GLSL are available in Godot's shading language.

Shader programs

In GLSL, each shader uses a separate program. You have one program for the vertex shader and one for the fragment shader. In Godot, you have a single shader that contains a `vertex` and/or a `fragment` function. If you only choose to write one, Godot will supply the other.

Godot allows uniform variables and functions to be shared by defining the fragment and vertex shaders in one file. In GLSL, the vertex and fragment programs cannot share variables except when varyings are used.

Vertex attributes

In GLSL, you can pass in per-vertex information using attributes and have the flexibility to pass in as much or as little as you want. In Godot, you have a set number of input attributes, including `VERTEX` (position), `COLOR`, `UV`, `UV2`, `NORMAL`. Each shaders' page in the shader reference section of the documentation comes with a complete list of its vertex attributes.

gl_Position

`gl_Position` receives the final position of a vertex specified in the vertex shader. It is specified by the user in clip space. Typically, in GLSL, the model space vertex position is passed in using a vertex attribute called `position` and you handle the conversion from model space to clip space manually.

In Godot, `VERTEX` specifies the vertex position in model space at the beginning of the `vertex` function. Godot also handles the final conversion to clip space after the user-defined `vertex` function is run. If you want to skip the conversion from model to view space, you can set the `render_mode` to `skip_vertex_transform`. If you want to skip all transforms, set `render_mode` to `skip_vertex_transform` and set the `PROJECTION_MATRIX` to `mat4(1.0)` in order to nullify the final transform from view space to clip space.

Varyings

Varyings are a type of variable that can be passed from the vertex shader to the fragment shader. In modern GLSL (3.0 and up), varyings are defined with the `in` and `out` keywords. A variable going out of the vertex shader is defined with `out` in the vertex shader and `in` inside the fragment shader.

Main

In GLSL, each shader program looks like a self-contained C-style program. Accordingly, the main entry point is `main`. If you are copying a vertex shader, rename `main` to `vertex` and if you are copying a fragment shader, rename `main` to `fragment`.

Macros

The Godot shader preprocessor supports the following macros:

- `#define` / `#undef`
- `#if`, `#elif`, `#else`, `#endif`, `defined()`, `#ifdef`, `#ifndef`
- `#include` (only `.gdshaderinc` files and with a maximum depth of 25)
- `#pragma disable_preprocessor`, which disables preprocessing for the rest of the file

Variables

GLSL has many built-in variables that are hard-coded. These variables are not uniforms, so they are not editable from the main program.

Variable	Type	Equivalent	Description
gl_FragColor	out vec4	COLOR	Output color for each pixel.
gl_FragCoord	vec4	FRAGCOORD	For full screen quads. For smaller quads, use UV.
gl_Position	vec4	VERTEX	Position of Vertex, output from Vertex Shader.
gl_PointSize	float	POINT_SIZE	Size of Point primitive.
gl_PointCoord	vec2	POINT_COORD	Position on point when drawing Point primitives.

Variable	Type	Equivalent	Description
gl_FrontFacing	bool	FRONT_FACING	True if front face of primitive.

Coordinates

`gl_FragCoord` in GLSL and `FRAGCOORD` in the Godot shading language use the same coordinate system. If using UV in Godot, the y-coordinate will be flipped upside down.

Precision

In GLSL, you can define the precision of a given type (float or int) at the top of the shader with the `precision` keyword. In Godot, you can set the precision of individual variables as you need by placing precision qualifiers `lowp`, `mediump`, and `highp` before the type when defining the variable. For more information, see the Shading Language reference.

Shadertoy

Shadertoy [https://www.shadertoy.com/results?query=&sort=popular&from=10&num=4] is a website that makes it easy to write fragment shaders and create pure magic [https://www.shadertoy.com/view/4tjGRh].

Shadertoy does not give the user full control over the shader. It handles all the input and uniforms and only lets the user write the fragment shader.

Types

Shadertoy uses the webgl spec, so it runs a slightly different version of GLSL. However, it still has the regular types, including constants and macros.

mainImage

The main point of entry to a Shadertoy shader is the `mainImage` function. `mainImage` has two parameters, `fragColor` and `fragCoord`, which correspond to `COLOR` and `FRAGCOORD` in Godot, respectively. These parameters are handled automatically in Godot, so you do not need to include them as parameters yourself. Anything in the `mainImage` function should be copied into the `fragment` function when porting to Godot.

Variables

In order to make writing fragment shaders straightforward and easy, Shadertoy handles passing a lot of helpful information from the main program into the fragment shader for you. A few of these have no equivalents in Godot because Godot has chosen not to make them available by default. This is okay because Godot gives you the ability to make your own uniforms. For variables whose equivalents are listed as "Provide with Uniform", users are responsible for creating that uniform themselves. The description gives the reader a hint about what they can pass in as a substitute.

Variable	Type	Equivalent	Description
fragColor	out vec4	COLOR	Output color for each pixel.
fragCoord	vec2	FRAGCOORD.xy	For full screen quads. For smaller

Variable	Type	Equivalent	Description
			quads, use UV.
iResolution	vec3	1.0 / SCREEN_PIXEL_SIZE	Can also pass in manually.
iTime	float	TIME	Time since shader started.
iTimeDelta	float	Provide with Uniform	Time to render previous frame.
iFrame	float	Provide with Uniform	Frame number.
iChannelTime[4]	float	Provide with Uniform	Time since that particular texture started.
iMouse	vec4	Provide with Uniform	Mouse position in pixel coordinates.

Variable	Type	Equivalent	Description
iDate	vec4	Provide with Uniform	Current date, expressed in seconds.
iChannelResolution[4]	vec3	1.0 / TEXTURE_PIXEL_SIZE	Resolution of particular texture.
iChanneli	Sampler2D	TEXTURE	Godot provides only one built-in; user can make more.

Coordinates

`fragCoord` behaves the same as `gl_FragCoord` in GLSL and `FRAGCOORD` in Godot.

The Book of Shaders

Similar to Shadertoy, The Book of Shaders [https://thebookofshaders.com] provides access to a fragment shader in the web browser, with which the user may interact. The user is restricted to writing fragment shader code with a set list of uniforms passed in and with no ability to add additional uniforms.

For further help on porting shaders to various frameworks generally, The Book of Shaders provides a page [https://thebookofshaders.com/04] on running shaders in various frameworks.

Types

The Book of Shaders uses the webgl spec, so it runs a slightly different version of GLSL. However, it still has the regular types, including constants and macros.

Main

The entry point for a Book of Shaders fragment shader is `main`, just like in GLSL. Everything written in a Book of Shaders `main` function should be copied into Godot's `fragment` function.

Variables

The Book of Shaders sticks closer to plain GLSL than Shadertoy does. It also implements fewer uniforms than Shadertoy.

Variable	Type	Equivalent	Description
gl_FragColor	out vec4	COLOR	Output color for each pixel.

Variable	**Type**	**Equivalent**	**Description**
gl_FragCoord	vec4	FRAGCOORD	For full screen quads. For smaller quads, use UV.
u_resolution	vec2	1.0 / SCREEN_PIXEL_SIZE	Can also pass in manually.
u_time	float	TIME	Time since shader started.
u_mouse	vec2	Provide with Uniform	Mouse position in pixel coordinates.

Coordinates

The Book of Shaders uses the same coordinate system as GLSL.

Shaders style guide

This style guide lists conventions to write elegant shaders. The goal is to encourage writing clean, readable code and promote consistency across projects, discussions, and tutorials. Hopefully, this will also support the development of auto-formatting tools.

Since the Godot shader language is close to C-style languages and GLSL, this guide is inspired by Godot's own GLSL formatting. You can view examples of GLSL files in Godot's source code here [https://github.com/godotengine/godot/blob/master/drivers/gles3/shaders/].

Style guides aren't meant as hard rulebooks. At times, you may not be able to apply some of the guidelines below. When that happens, use your best judgment, and ask fellow developers for insights.

In general, keeping your code consistent in your projects and within your team is more important than following this guide to a tee.

Note

Godot's built-in shader editor uses a lot of these conventions by default. Let it help you.

Here is a complete shader example based on these guidelines:

```
shader_type canvas_item;
// Screen-space shader to adjust a 2D scene's brightness,
contrast
// and saturation. Taken from
// https://github.com/godotengine/godot-demo-
projects/blob/master/2d/screen_space_shaders/shaders/BCS.gdshad
er

uniform sampler2D screen_texture : hint_screen_texture,
filter_linear_mipmap;
uniform float brightness = 0.8;
uniform float contrast = 1.5;
uniform float saturation = 1.8;

void fragment() {
    vec3 c = textureLod(screen_texture, SCREEN_UV, 0.0).rgb;

    c.rgb = mix(vec3(0.0), c.rgb, brightness);
    c.rgb = mix(vec3(0.5), c.rgb, contrast);
    c.rgb = mix(vec3(dot(vec3(1.0), c.rgb) * 0.33333), c.rgb,
saturation);

    COLOR.rgb = c;
}
```

Formatting

Encoding and special characters

- Use line feed (**LF**) characters to break lines, not CRLF or CR. *(editor default)*
- Use one line feed character at the end of each file. *(editor default)*
- Use **UTF-8** encoding without a byte order mark [https://en.wikipedia.org/wiki/Byte_order_mark]. *(editor default)*
- Use **Tabs** instead of spaces for indentation. *(editor default)*

Indentation

Each indent level should be one tab greater than the block containing it.

Good:

```
void fragment() {
    COLOR = vec3(1.0, 1.0, 1.0);
}
```

Bad:

```
void fragment() {
        COLOR = vec3(1.0, 1.0, 1.0);
}
```

Use 2 indent levels to distinguish continuation lines from regular code blocks.

Good:

```
vec2 st = vec2(
        atan(NORMAL.x, NORMAL.z),
        acos(NORMAL.y));
```

Bad:

```
vec2 st = vec2(
    atan(NORMAL.x, NORMAL.z),
    acos(NORMAL.y));
```

Line breaks and blank lines

For a general indentation rule, follow [the "1TBS Style"](https://en.wikipedia.org/wiki/Indentation_style#Variant:_1TBS_(OTBS)) [https://en.wikipedia.org/wiki/Indentation_style#Variant:_1TBS_(OTBS)] which recommends placing the brace associated with a control statement on the same line. Always use braces for statements, even if they only span one line. This makes them easier to refactor and avoids mistakes when adding more lines to an `if` statement or similar.

Good:

```
void fragment() {
    if (true) {
        // ...
    }
}
```

Bad:

```
void fragment()
{
    if (true)
        // ...
}
```

Blank lines

Surround function definitions with one (and only one) blank line:

```
void do_something() {
    // ...
}

void fragment() {
    // ...
}
```

Use one (and only one) blank line inside functions to separate logical sections.

Line length

Keep individual lines of code under 100 characters.

If you can, try to keep lines under 80 characters. This helps to read the code on small displays and with two shaders opened side-by-side in an external text editor. For example, when looking at a differential revision.

One statement per line

Never combine multiple statements on a single line.

Good:

```
void fragment() {
    ALBEDO = vec3(1.0);
    EMISSION = vec3(1.0);
}
```

Bad:

```
void fragment() {
    ALBEDO = vec3(1.0); EMISSION = vec3(1.0);
}
```

The only exception to that rule is the ternary operator:

```
void fragment() {
     bool should_be_white = true;
     ALBEDO = should_be_white ? vec3(1.0) : vec3(0.0);
 }
```

Comment spacing

Regular comments should start with a space, but not code that you comment out. This helps differentiate text comments from disabled code.

Good:

```
// This is a comment.
//return;
```

Bad:

```
//This is a comment.
// return;
```

Don't use multiline comment syntax if your comment can fit on a single line:

```
/* This is another comment. */
```

Note

In the shader editor, to make the selected code a comment (or uncomment it), press `Ctrl + K`. This feature adds or removes `//` at the start of the selected lines.

Documentation comments

Use the following format for documentation comments above uniforms, with **two** leading asterisks (`/**`) and follow-up asterisks on every line:

```
/**
 * This is a documentation comment.
 * These lines will appear in the inspector when hovering the
shader parameter
 * named "Something".
 * You can use [b]BBCode[/b] [i]formatting[/i] in the comment.
 */
uniform int something = 1;
```

These comments will appear when hovering a property in the inspector. If you don't wish the comment to be visible in the inspector, use the standard comment syntax instead (`// ...` or `/* ... */` with only one leading asterisk).

Whitespace

Always use one space around operators and after commas. Also, avoid extraneous spaces in function calls.

Good:

```
COLOR.r = 5.0;
COLOR.r = COLOR.g + 0.1;
COLOR.b = some_function(1.0, 2.0);
```

Bad:

```
COLOR.r=5.0;
COLOR.r = COLOR.g+0.1;
COLOR.b = some_function (1.0,2.0);
```

Don't use spaces to align expressions vertically:

```
ALBEDO.r   = 1.0;
EMISSION.r = 1.0;
```

Floating-point numbers

Always specify at least one digit for both the integer and fractional part. This makes it easier to distinguish floating-point numbers from integers, as well as distinguishing numbers greater than 1 from those lower than 1.

Good:

```
void fragment() {
    ALBEDO.rgb = vec3(5.0, 0.1, 0.2);
}
```

Bad:

```
void fragment() {
    ALBEDO.rgb = vec3(5., .1, .2);
}
```

Accessing vector members

Use `r`, `g`, `b`, and `a` when accessing a vector's members if it contains a color. If the vector contains anything else than a color, use `x`, `y`, `z`, and `w`. This allows those reading your code to better understand what the underlying data represents.

Good:

```
COLOR.rgb = vec3(5.0, 0.1, 0.2);
```

Bad:

```
COLOR.xyz = vec3(5.0, 0.1, 0.2);
```

Naming conventions

These naming conventions follow the Godot Engine style. Breaking these will make your code clash with the built-in naming conventions, leading to inconsistent code.

Functions and variables

Use snake_case to name functions and variables:

```
void some_function() {
     float some_variable = 0.5;
}
```

Constants

Write constants with CONSTANT_CASE, that is to say in all caps with an underscore (_) to separate words:

```
const float GOLDEN_RATIO = 1.618;
```

Preprocessor directives

Shader preprocessor directives should be written in CONSTANT__CASE. Directives should be written without any indentation before them, even if nested within a function.

To preserve the natural flow of indentation when shader errors are printed to the console, extra indentation should **not** be added within `#if`, `#ifdef` or `#ifndef` blocks:

Good:

```
#define HEIGHTMAP_ENABLED

void fragment() {
    vec2 position = vec2(1.0, 2.0);

#ifdef HEIGHTMAP_ENABLED
    sample_heightmap(position);
#endif
}
```

Bad:

```
#define heightmap_enabled

void fragment() {
    vec2 position = vec2(1.0, 2.0);

    #ifdef heightmap_enabled
        sample_heightmap(position);
    #endif
}
```

Code order

We suggest to organize shader code this way:

```
01. shader type declaration
02. render mode declaration
03. // docstring
```

```
uniforms
constants
varyings

other functions
vertex() function
fragment() function
light() function
```

We optimized the order to make it easy to read the code from top to bottom, to help developers reading the code for the first time understand how it works, and to avoid errors linked to the order of variable declarations.

This code order follows two rules of thumb:

1. Metadata and properties first, followed by methods.
2. "Public" comes before "private". In a shader language's context, "public" refers to what's easily adjustable by the user (uniforms).

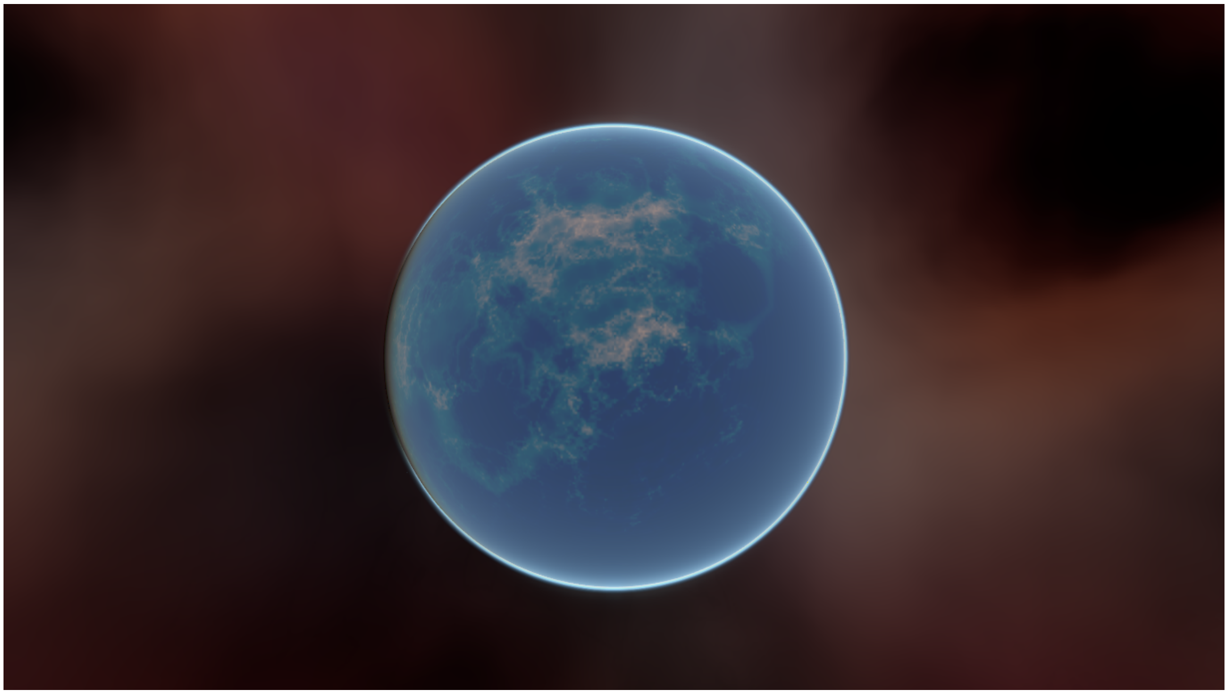
Local variables

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Using a SubViewport as a texture

Introduction

This tutorial will introduce you to using the SubViewport as a texture that can be applied to 3D objects. In order to do so, it will walk you through the process of making a procedural planet like the one below:



Note

This tutorial does not cover how to code a dynamic atmosphere like the one this planet has.

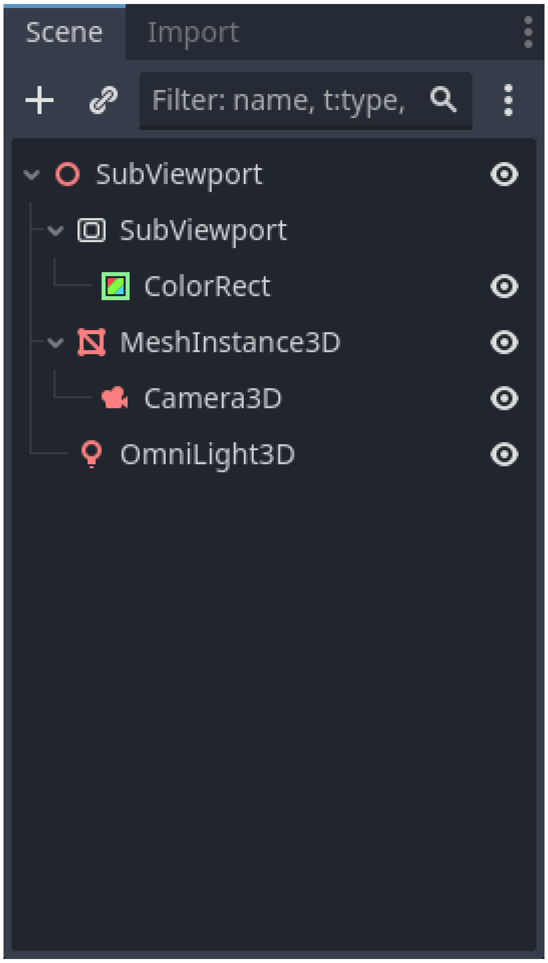
This tutorial assumes you are familiar with how to set up a basic scene including: a Camera3D, a light source, a MeshInstance3D with a Primitive Mesh, and applying a StandardMaterial3D to the mesh. The focus will be on using the SubViewport to dynamically create textures that can be applied to the mesh.

In this tutorial, we'll cover the following topics:

- How to use a SubViewport as a render texture
- Mapping a texture to a sphere with equirectangular mapping
- Fragment shader techniques for procedural planets
- Setting a Roughness map from a Viewport Texture

Setting up the scene

Create a new scene and add the following nodes exactly as shown below.



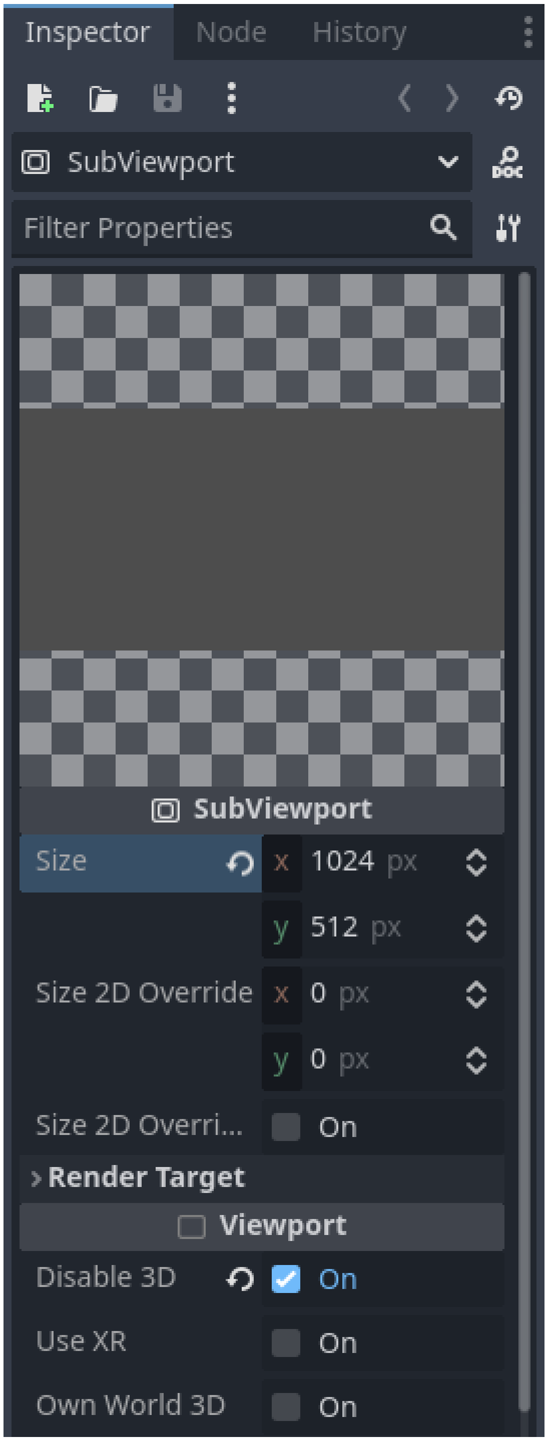
SubViewport
- SubViewport
  - ColorRect
- MeshInstance3D
  - Camera3D
- OmniLight3D

Go into the the MeshInstance3D and make the mesh a SphereMesh

Setting up the SubViewport

Click on the SubViewport node and set its size to `(1024, 512)`. The SubViewport can actually be any size so long as the width is double the height. The width needs to be double the height so that the image will accurately map onto the sphere, as we will be using equirectangular projection, but more on that later.

Next disable 3D. We will be using a ColorRect to render the surface, so we don't need 3D either.


Inspector
Node
History
SubViewport
Filter Properties
SubViewport

Size x 1024 px y 512 px

Size 2D Override x 0 px y 0 px

Size 2D Overri... ☐ On

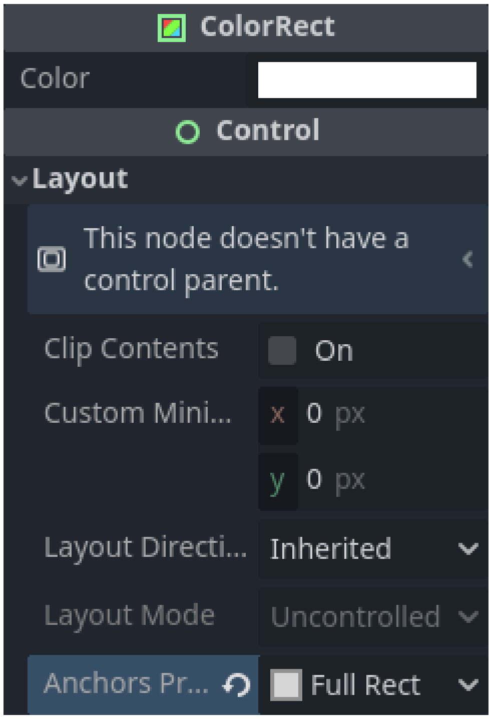
Render Target
Viewport

Disable 3D ☑ On

Use XR ☐ On

Own World 3D ☐ On

Select the ColorRect and in the inspector set the anchors preset to `Full Rect`. This will ensure that the ColorRect takes up the entire SubViewport.



This node doesn't have a control parent.

Clip Contents: ☐ On

Custom Mini...: x 0 px, y 0 px

Layout Directi...: Inherited

Layout Mode: Uncontrolled

Anchors Pr...: Full Rect

Next, we add a Shader Material to the ColorRect (ColorRect > CanvasItem > Material > Material > `New ShaderMaterial`).

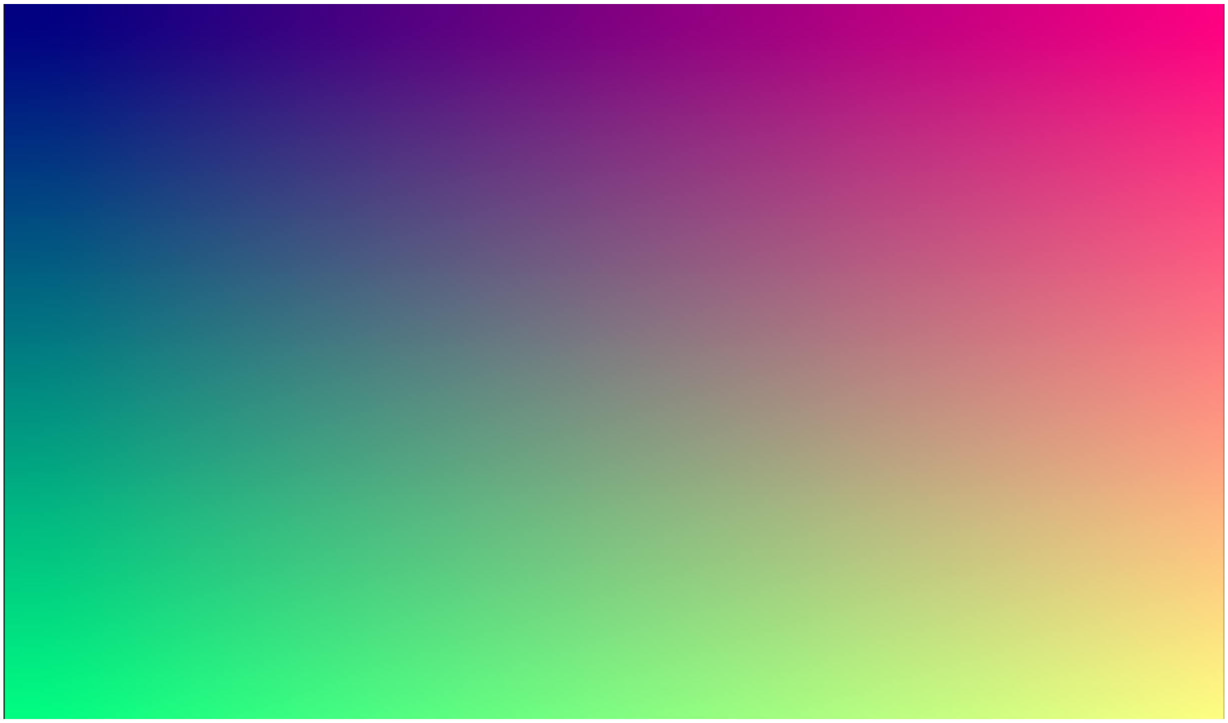
Note

Basic familiarity with shading is recommended for this tutorial. However, even if you are new to shaders, all the code will be provided, so you should have no problem following along.

Click the dropdown menu button for the shader material and click / Edit. From here go to Shader > `New Shader`. give it a name and click "Create". click the shader in the inspector to open the shader editor. Delete the default code and add the following:

```
shader_type canvas_item;

void fragment() {
    COLOR = vec4(UV.x, UV.y, 0.5, 1.0);
}
```

save the shader code, you'll see in the inspector that the above code renders a gradient like the one below.



Now we have the basics of a SubViewport that we render to and we have a unique image that we can apply to the sphere.

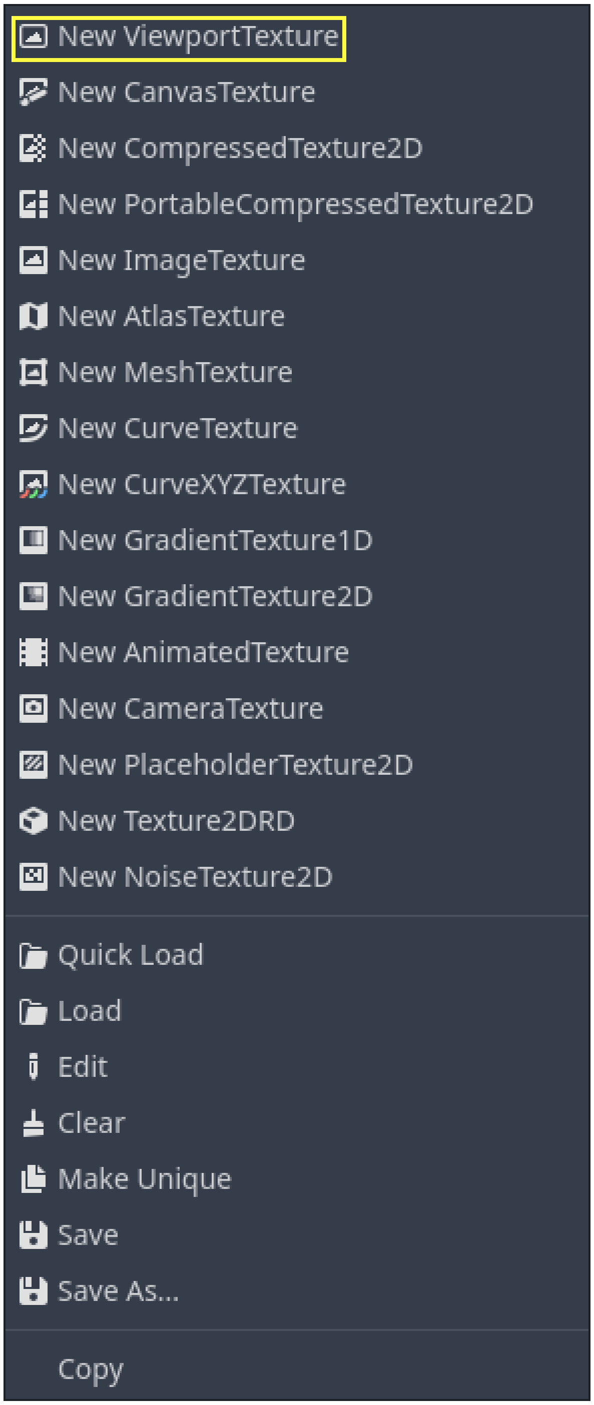
Applying the texture

Now go into the MeshInstance3D and add a StandardMaterial3D to it. No need for a special Shader Material (although that would be a good idea for more advanced effects, like the atmosphere in the example above).

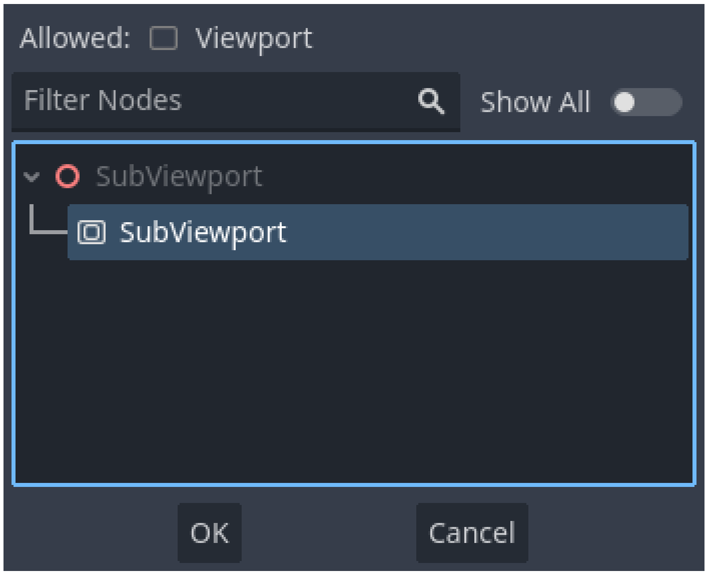
MeshInstance3D > GeometryInstance > Geometry > Material Override > `New StandardMaterial3D`

Then click the dropdown for the StandardMaterial3D and click "Edit"

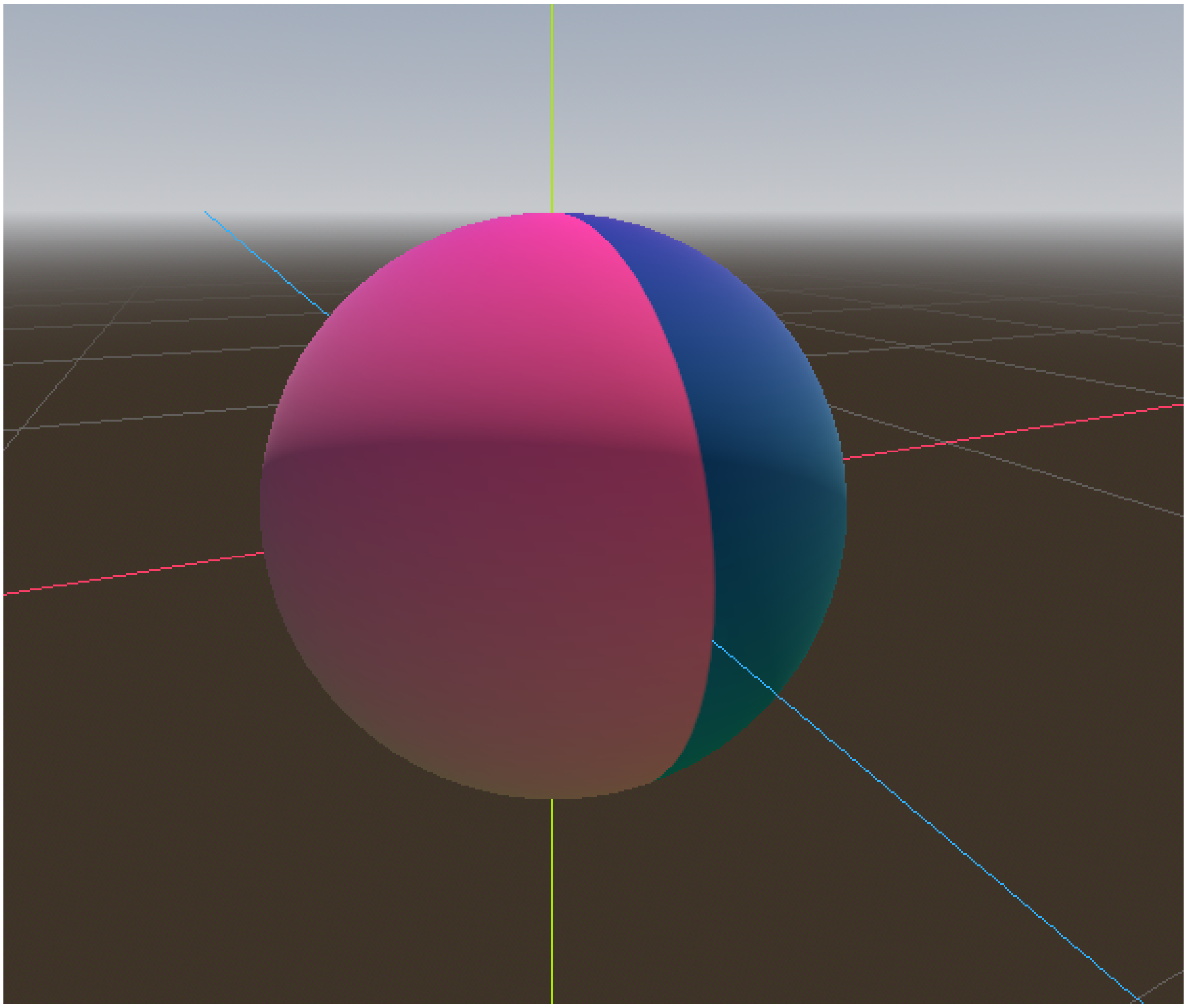
Go to the "Resource" section and check the `Local to scene` box. Then, go to the "Albedo" section and click beside the "Texture" property to add an Albedo Texture. Here we will apply the texture we made. Choose "New ViewportTexture"


New ViewportTexture
New CanvasTexture
New CompressedTexture2D
New PortableCompressedTexture2D
New ImageTexture
New AtlasTexture
New MeshTexture
New CurveTexture
New CurveXYZTexture
New GradientTexture1D
New GradientTexture2D
New AnimatedTexture
New CameraTexture
New PlaceholderTexture2D
New Texture2DRD
New NoiseTexture2D
Quick Load
Load
Edit
Clear
Make Unique
Save
Save As...
Copy

Click on the ViewportTexture you just created in the inspector, then click "Assign". Then, from the menu that pops up, select the Viewport that we rendered to earlier.


Allowed: ☐ Viewport
Filter Nodes
Show All
SubViewport
SubViewport
OK
Cancel

Your sphere should now be colored in with the colors we rendered to the Viewport.

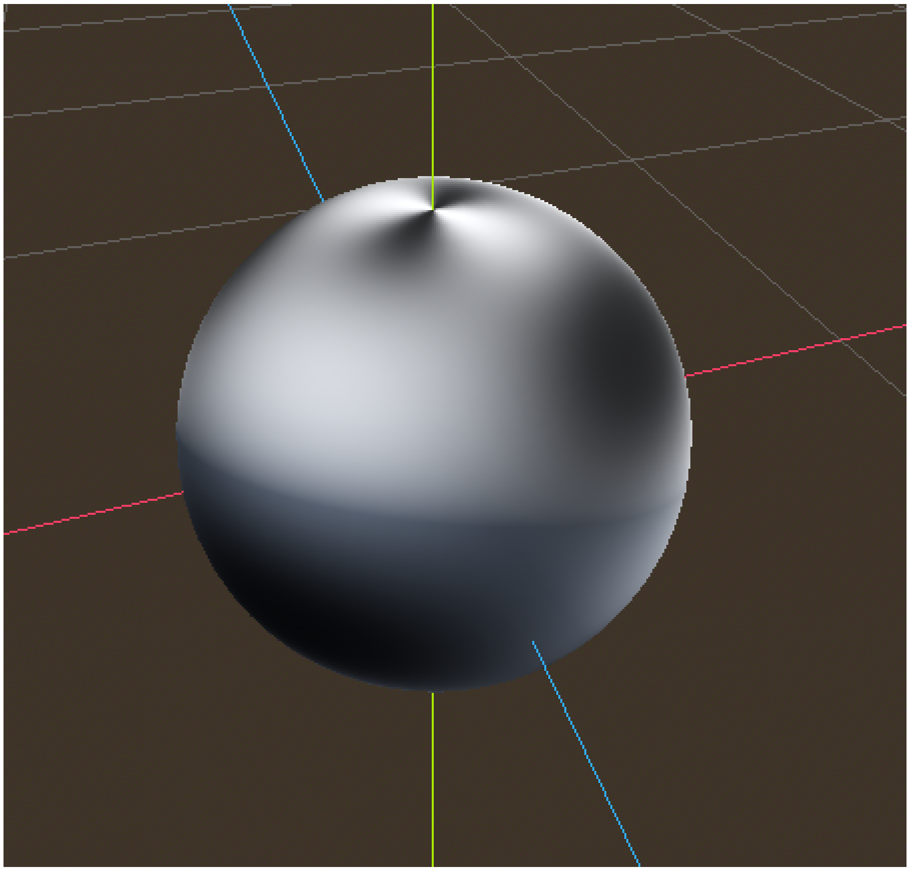


Notice the ugly seam that forms where the texture wraps around? This is because we are picking a color based on UV coordinates and UV coordinates do not wrap around the texture. This is a classic problem in 2D map projection. Game developers often have a 2-dimensional map they want to project onto a sphere, but when it wraps around, it has large seams. There is an elegant workaround for this problem that we will illustrate in the next section.

Making the planet texture

So now, when we render to our SubViewport, it appears magically on the sphere. But there is an ugly seam created by our texture coordinates. So how do we get a range of coordinates that wrap around the sphere in a nice way? One solution is to use a function that repeats on the domain of our texture. `sin` and `cos` are two such functions. Let's apply them to the texture and see what happens. Replace the existing color code in the shader with the following:

```
COLOR.xyz = vec3(sin(UV.x * 3.14159 * 4.0) * cos(UV.y * 3.14159
* 4.0) * 0.5 + 0.5);
```



Not too bad. If you look around, you can see that the seam has now disappeared, but in its place, we have pinching at the poles. This pinching is due to the way Godot maps textures to spheres in its StandardMaterial3D. It uses a projection technique called equirectangular projection, which translates a spherical map onto a 2D plane.

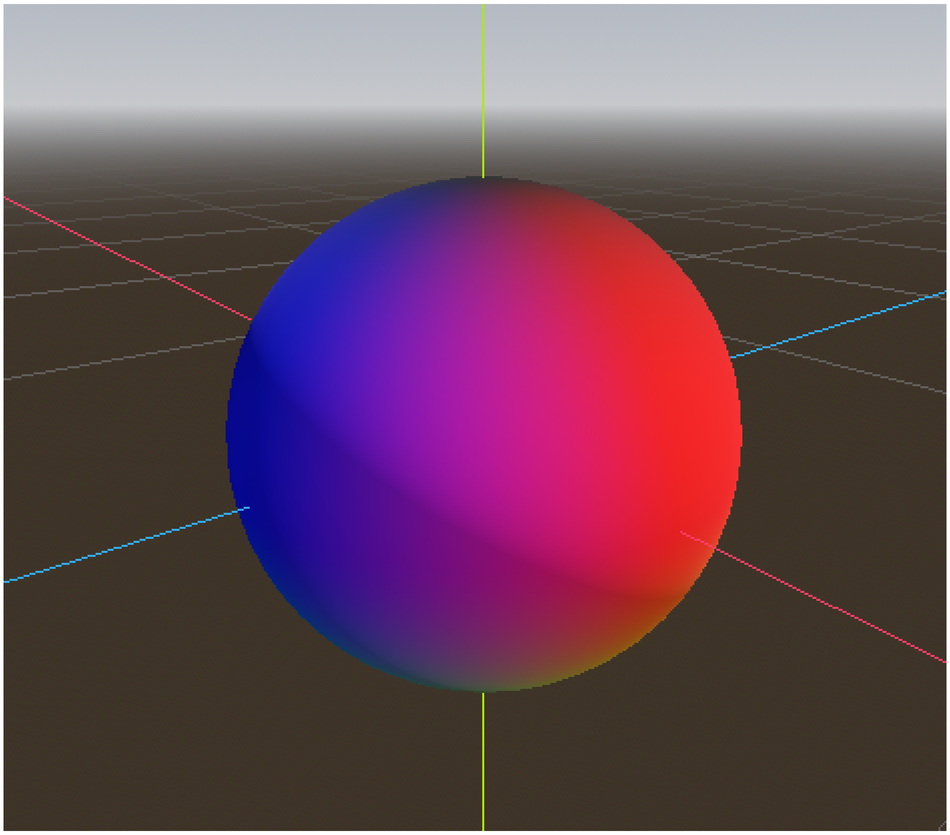
Note

If you are interested in a little extra information on the technique, we will be converting from spherical coordinates into Cartesian coordinates. Spherical coordinates map the longitude and latitude of the sphere, while Cartesian coordinates are, for all intents and purposes, a vector from the center of the sphere to the point.

For each pixel, we will calculate its 3D position on the sphere. From that, we will use 3D noise to determine a color value. By calculating the noise in 3D, we solve the problem of the pinching at the poles. To understand why, picture the noise being calculated across the surface of the sphere instead of across the 2D plane. When you calculate across the surface of the sphere, you never hit an edge, and hence you never create a seam or a pinch point on the pole. The following code converts the `UVs` into Cartesian coordinates.

```
float theta = UV.y * 3.14159;
float phi = UV.x * 3.14159 * 2.0;
vec3 unit = vec3(0.0, 0.0, 0.0);

unit.x = sin(phi) * sin(theta);
unit.y = cos(theta) * -1.0;
unit.z = cos(phi) * sin(theta);
unit = normalize(unit);
```

And if we use `unit` as an output `COLOR` value, we get:



Now that we can calculate the 3D position of the surface of the sphere, we can use 3D noise to make the planet. We will be using this noise function directly from a Shadertoy [https://www.shadertoy.com/view/Xsl3Dl]:

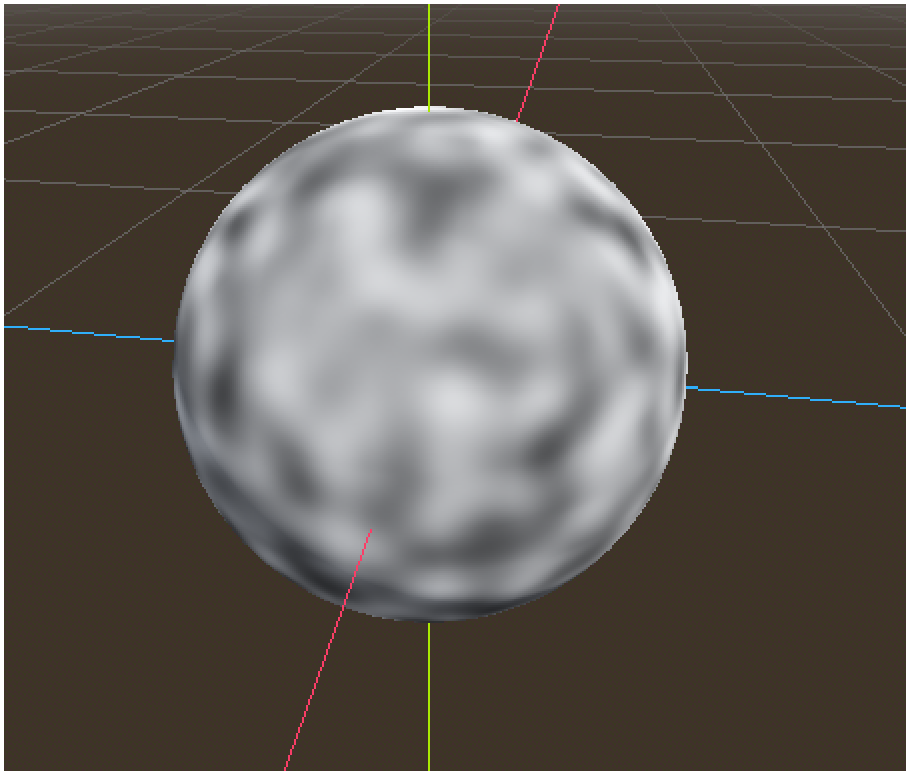
```
vec3 hash(vec3 p) {
    p = vec3(dot(p, vec3(127.1, 311.7, 74.7)),
             dot(p, vec3(269.5, 183.3, 246.1)),
             dot(p, vec3(113.5, 271.9, 124.6)));

    return -1.0 + 2.0 * fract(sin(p) * 43758.5453123);
}

float noise(vec3 p) {
  vec3 i = floor(p);
  vec3 f = fract(p);
  vec3 u = f * f * (3.0 - 2.0 * f);

  return mix(mix(mix(dot(hash(i + vec3(0.0, 0.0, 0.0)), f -
vec3(0.0, 0.0, 0.0)),
                     dot(hash(i + vec3(1.0, 0.0, 0.0)), f -
vec3(1.0, 0.0, 0.0)), u.x),
                 mix(dot(hash(i + vec3(0.0, 1.0, 0.0)), f -
vec3(0.0, 1.0, 0.0)),
                     dot(hash(i + vec3(1.0, 1.0, 0.0)), f -
vec3(1.0, 1.0, 0.0)), u.x), u.y),
             mix(mix(dot(hash(i + vec3(0.0, 0.0, 1.0)), f -
vec3(0.0, 0.0, 1.0)),
                     dot(hash(i + vec3(1.0, 0.0, 1.0)), f -
vec3(1.0, 0.0, 1.0)), u.x),
                 mix(dot(hash(i + vec3(0.0, 1.0, 1.0)), f -
vec3(0.0, 1.0, 1.0)),
                     dot(hash(i + vec3(1.0, 1.0, 1.0)), f -
vec3(1.0, 1.0, 1.0)), u.x), u.y), u.z );
}
```

Note

All credit goes to the author, Inigo Quilez. It is published under the `MIT` licence.

Now to use `noise`, add the following to the `fragment` function:

```
float n = noise(unit * 5.0);
COLOR.xyz = vec3(n * 0.5 + 0.5);
```



Note

In order to highlight the texture, we set the material to unshaded.

You can see now that the noise indeed wraps seamlessly around the sphere. Although this looks nothing like the planet you were promised. So let's move onto something more colorful.

Coloring the planet

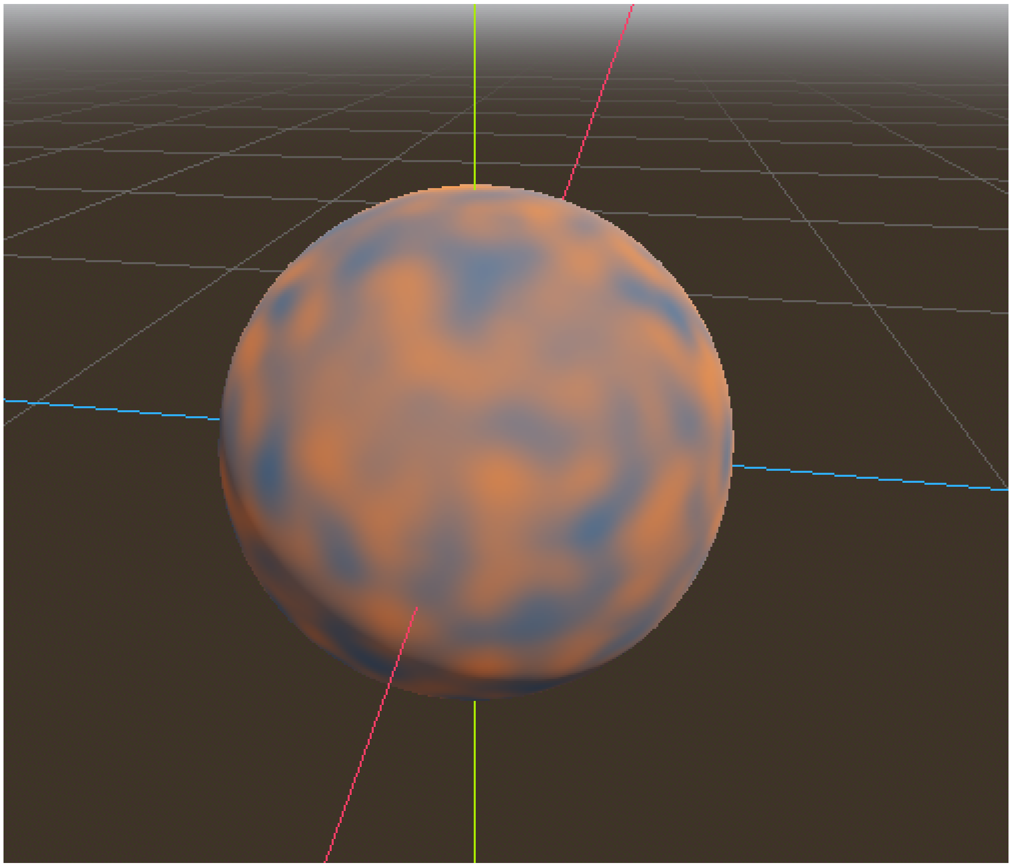
Now to make the planet colors. While there are many ways to do this, for now, we will stick with a gradient between water and land.

To make a gradient in GLSL, we use the `mix` function. `mix` takes two values to interpolate between and a third argument to choose how much to interpolate between them; in essence, it *mixes* the two values together. In other APIs, this function is often called `lerp`.

However, `lerp` is typically reserved for mixing two floats together; `mix` can take any values whether it be floats or vector types.

```
COLOR.xyz = mix(vec3(0.05, 0.3, 0.5), vec3(0.9, 0.4, 0.1), n *
0.5 + 0.5);
```

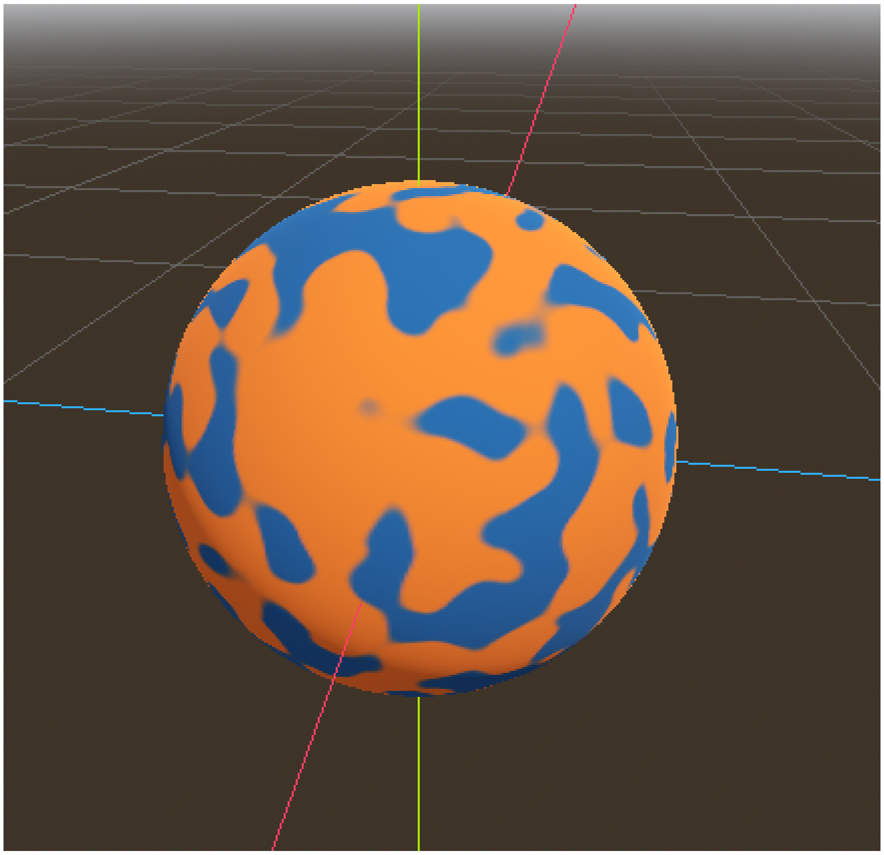
The first color is blue for the ocean. The second color is a kind of reddish color (because all alien planets need red terrain). And finally, they are mixed together by `n * 0.5 + 0.5`. `n` smoothly varies between `-1` and `1`. So we map it into the `0-1` range that `mix` expects. Now you can see that the colors change between blue and red.



That is a little more blurry than we want. Planets typically have a relatively clear separation between land and sea. In order to do that, we will change the last term to `smoothstep(-0.1, 0.0, n)`. And thus the whole line becomes:

```
COLOR.xyz = mix(vec3(0.05, 0.3, 0.5), vec3(0.9, 0.4, 0.1),
smoothstep(-0.1, 0.0, n));
```

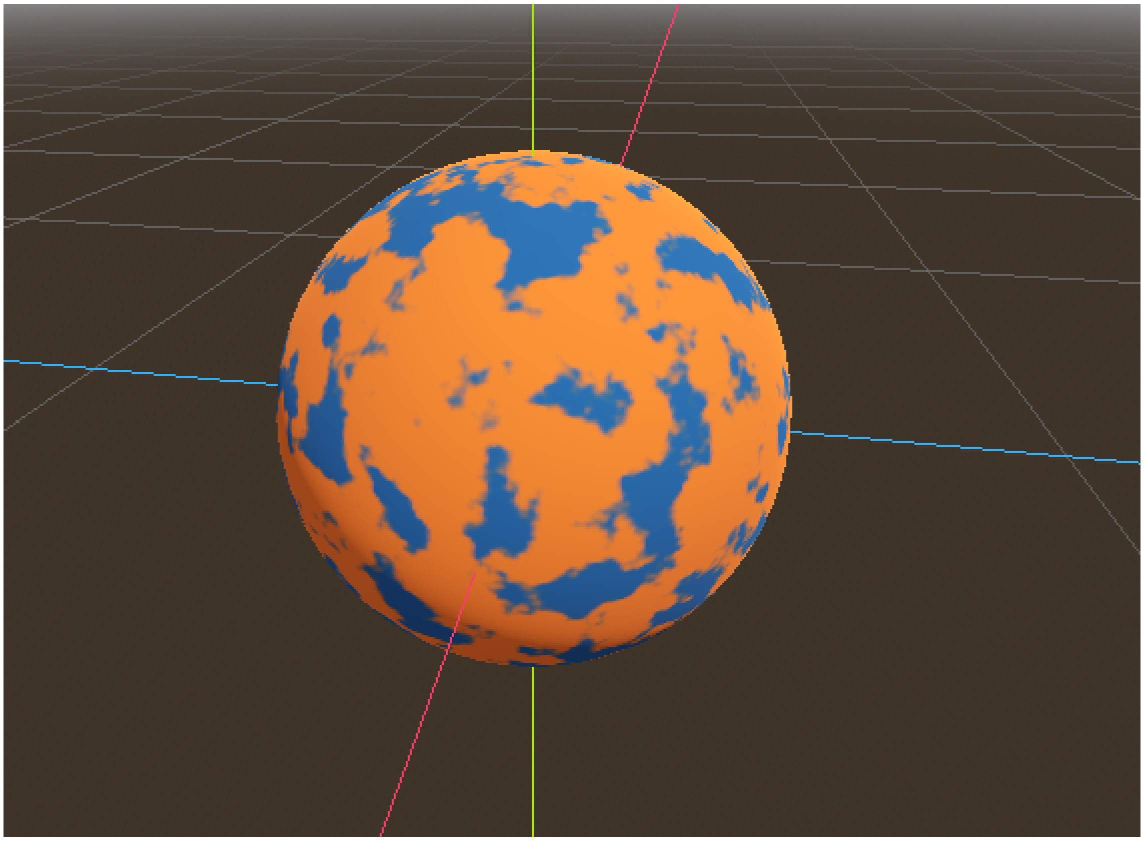
What `smoothstep` does is return `0` if the third argument is below the first and `1` if the third argument is larger than the second and smoothly blends between `0` and `1` if the third number is between the first and the second. So in this line, `smoothstep` returns `0` whenever `n` is less than `-0.1` and it returns `1` whenever `n` is above `0`.



One more thing to make this a little more planet-y. The land shouldn't be so blobby; let's make the edges a little rougher. A trick that is often used in shaders to make rough looking terrain with noise is to layer levels of noise over one another at various frequencies. We use one layer to make the overall blobby structure of the continents. Then another layer breaks up the edges a bit, and then another, and so on. What we will do is calculate `n` with four lines of shader code instead of just one. `n` becomes:

```
float n = noise(unit * 5.0) * 0.5;
n += noise(unit * 10.0) * 0.25;
n += noise(unit * 20.0) * 0.125;
n += noise(unit * 40.0) * 0.0625;
```

And now the planet looks like:



Making an ocean

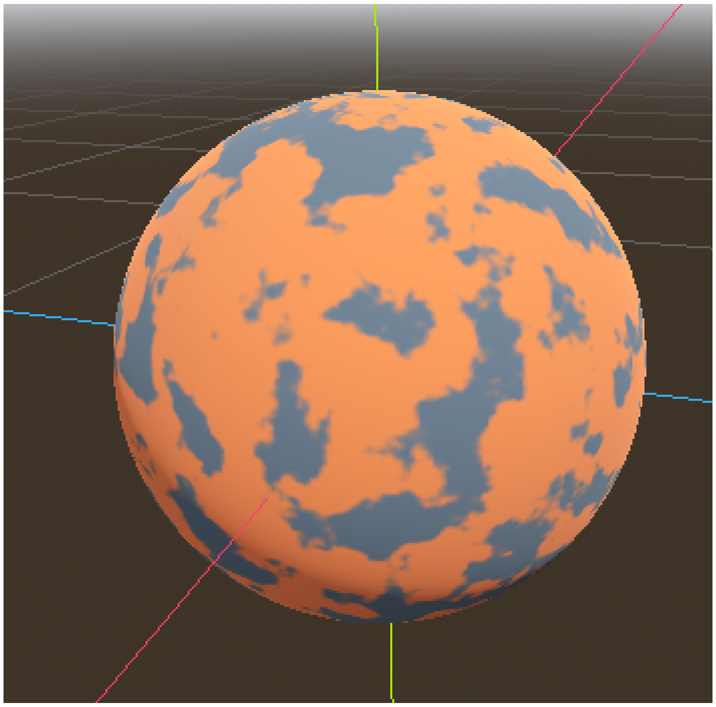
One final thing to make this look more like a planet. The ocean and the land reflect light differently. So we want the ocean to shine a little more than the land. We can do this by passing a fourth value into the `alpha` channel of our output `COLOR` and using it as a Roughness map.

```
COLOR.a = 0.3 + 0.7 * smoothstep(-0.1, 0.0, n);
```

This line returns `0.3` for water and `1.0` for land. This means that the land is going to be quite rough, while the water will be quite smooth.

And then, in the material, under the "Metallic" section, make sure `Metallic` is set to `0` and `Specular` is set to `1`. The reason for this is the water reflects light really well, but isn't metallic. These values are not physically accurate, but they are good enough for this demo.

Next, under the "Roughness" section set the roughness texture to a Viewport Texture pointing to our planet texture SubViewport. Finally, set the `Texture Channel` to `Alpha`. This instructs the renderer to use the `alpha` channel of our output `COLOR` as the `Roughness` value.

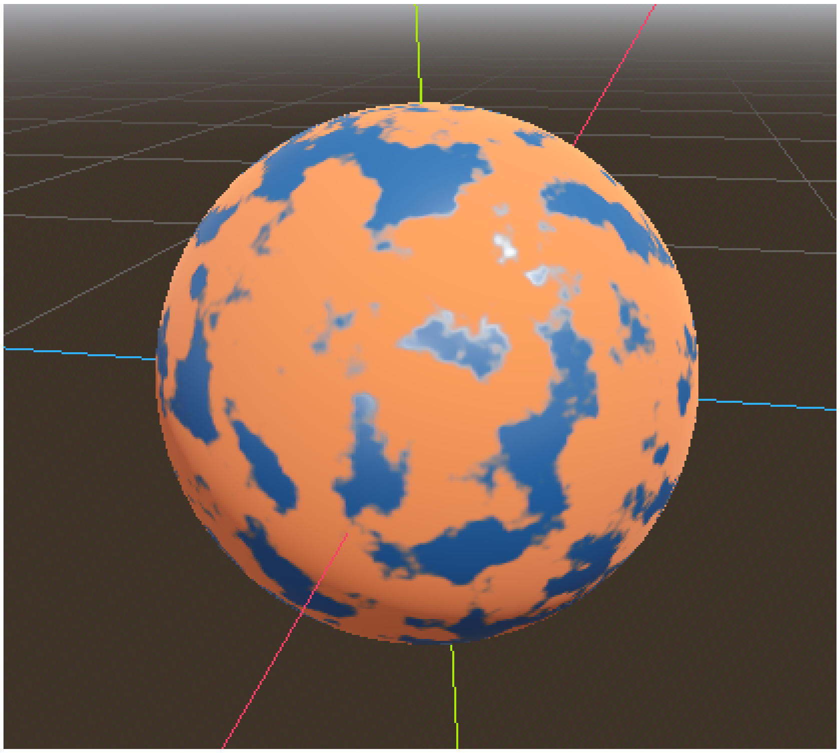


You'll notice that very little changes except that the planet is no longer reflecting the sky. This is happening because, by default, when something is rendered with an alpha value, it gets drawn as a transparent object over the background. And since the default background of the SubViewport is opaque, the `alpha` channel of the Viewport Texture is `1`, resulting in the planet texture being drawn with slightly fainter colors and a `Roughness` value of `1` everywhere. To correct this, we go into the SubViewport and enable the "Transparent Bg" property. Since we are now rendering one transparent object on top of another, we want to enable `blend_premul_alpha`:

```
render_mode blend_premul_alpha;
```

This pre-multiplies the colors by the `alpha` value and then blends them correctly together. Typically, when blending one transparent color on top of another, even if the background has an `alpha` of `0` (as it does in this case), you end up with weird color bleed issues. Setting `blend_premul_alpha` fixes that.

Now the planet should look like it is reflecting light on the ocean but not the land. move around the OmniLight3D in the scene so you can see the effect of the reflections on the ocean.



And there you have it. A procedural planet generated using a SubViewport.

Custom post-processing

Introduction

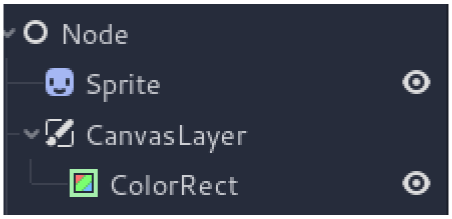
Godot provides many post-processing effects out of the box, including Bloom, DOF, and SSAO, which are described in Environment and post-processing. However, advanced use cases may require custom effects. This article explains how to write your own custom effects.

The easiest way to implement a custom post-processing shader is to use Godot's built-in ability to read from the screen texture. If you're not familiar with this, you should read the Screen Reading Shaders Tutorial first.

Single pass post-processing

Post-processing effects are shaders applied to a frame after Godot has rendered it. To apply a shader to a frame, create a CanvasLayer, and give it a ColorRect. Assign a new ShaderMaterial to the newly created `ColorRect`, and set the `ColorRect`'s layout to "Full Rect".

Your scene tree will look something like this:



Note

Another more efficient method is to use a BackBufferCopy to copy a region of the screen to a buffer and to access it in a shader script through a `sampler2D` using `hint_screen_texture`.

Note

As of the time of writing, Godot does not support rendering to multiple buffers at the same time. Your post-processing shader will not have access to other render passes and buffers not exposed by Godot (such as depth or normal/roughness). You only have access to the rendered frame and buffers exposed by Godot as samplers.

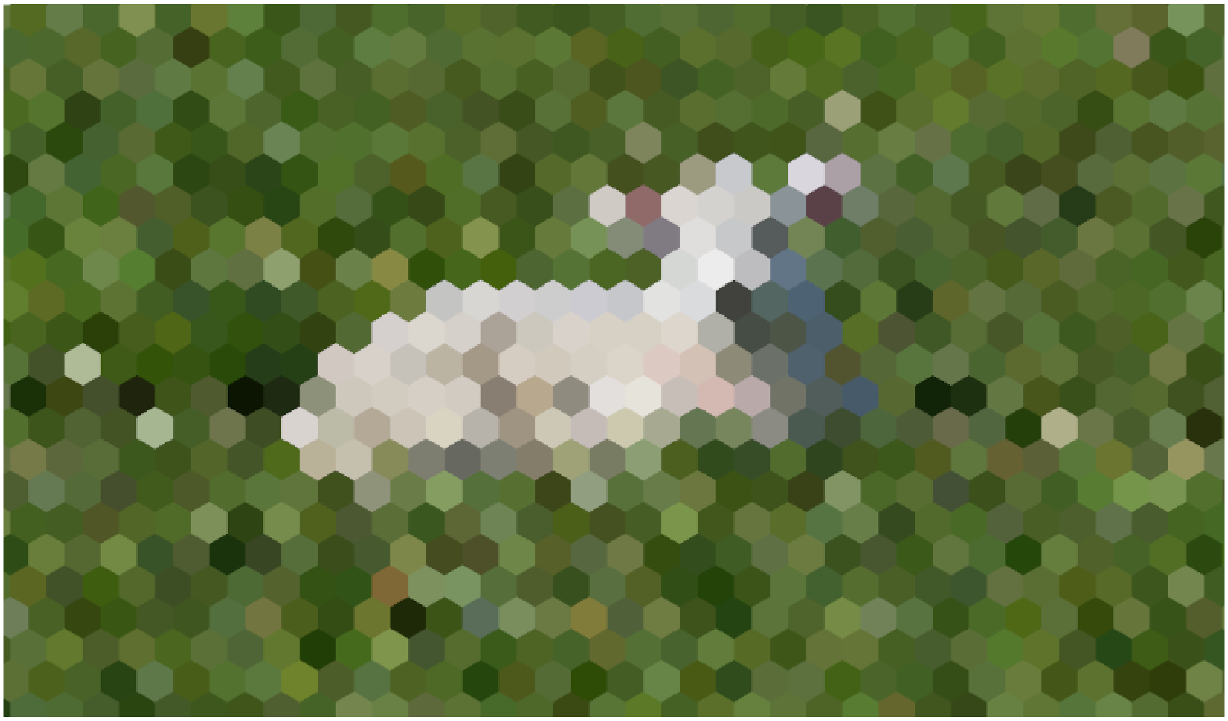
For this demo, we will use this Sprite of a sheep.

Assign a new Shader to the `ColorRect`'s `ShaderMaterial`. You can access the frame's texture and UV with a `sampler2D` using `hint_screen_texture` and the built in `SCREEN_UV` uniforms.

Copy the following code to your shader. The code below is a hex pixelization shader by arlez80 [https://bitbucket.org/arlez80/hex-mosaic/src/master/],

```
shader_type canvas_item;

uniform vec2 size = vec2(32.0, 28.0);
// If you intend to read from mipmaps with `textureLod()` LOD values greater than `0.0`,
// use `filter_nearest_mipmap` instead. This shader doesn't require it.
uniform sampler2D screen_texture : hint_screen_texture, repeat_disable, filter_nearest;

void fragment() {
        vec2 norm_size = size * SCREEN_PIXEL_SIZE;
        bool half = mod(SCREEN_UV.y / 2.0, norm_size.y) / norm_size.y < 0.5;
        vec2 uv = SCREEN_UV + vec2(norm_size.x * 0.5 * float(half), 0.0);
        vec2 center_uv = floor(uv / norm_size) * norm_size;
        vec2 norm_uv = mod(uv, norm_size) / norm_size;
        center_uv += mix(vec2(0.0, 0.0),
                         mix(mix(vec2(norm_size.x, -norm_size.y),
                                 vec2(0.0, -norm_size.y),
                                 float(norm_uv.x < 0.5)),
                             mix(vec2(0.0, -norm_size.y),
                                 vec2(-norm_size.x, -norm_size.y),
                                 float(norm_uv.x < 0.5)),
                             float(half)),
                         float(norm_uv.y < 0.3333333) * float(norm_uv.y / 0.3333333 < (abs(norm_uv.x - 0.5) * 2.0)));

        COLOR = textureLod(screen_texture, center_uv, 0.0);
}
```

The sheep will look something like this:

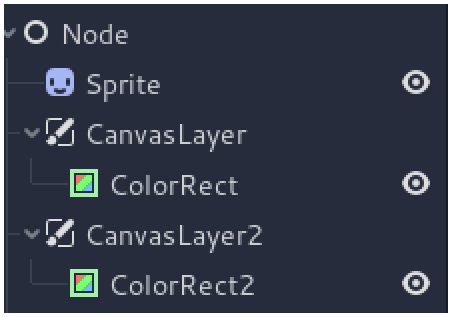


Multi-pass post-processing

Some post-processing effects like blurs are resource intensive. You can make them run a lot faster if you break them down in multiple passes. In a multipass material, each pass takes the result from the previous pass as an input and processes it.

To produce a multi-pass post-processing shader, you stack `CanvasLayer` and `ColorRect` nodes. In the example above, you use a `CanvasLayer` object to render a shader using the frame on the layer below. Apart from the node structure, the steps are the same as with the single-pass post-processing shader.

Your scene tree will look something like this:



As an example, you could write a full screen Gaussian blur effect by attaching the following pieces of code to each of the `ColorRect` nodes. The order in which you apply the shaders depends on the position of the `CanvasLayer` in the scene tree, higher means sooner. For this blur shader, the order does not matter.

```
shader_type canvas_item;

uniform sampler2D screen_texture : hint_screen_texture,
repeat_disable, filter_nearest;

// Blurs the screen in the X-direction.
void fragment() {
    vec3 col = texture(screen_texture, SCREEN_UV).xyz * 0.16;
    col += texture(screen_texture, SCREEN_UV +
vec2(SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.15;
    col += texture(screen_texture, SCREEN_UV + vec2(-
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.15;
    col += texture(screen_texture, SCREEN_UV + vec2(2.0 *
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.12;
    col += texture(screen_texture, SCREEN_UV + vec2(2.0 * -
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.12;
    col += texture(screen_texture, SCREEN_UV + vec2(3.0 *
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.09;
    col += texture(screen_texture, SCREEN_UV + vec2(3.0 * -
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.09;
    col += texture(screen_texture, SCREEN_UV + vec2(4.0 *
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.05;
    col += texture(screen_texture, SCREEN_UV + vec2(4.0 * -
SCREEN_PIXEL_SIZE.x, 0.0)).xyz * 0.05;
    COLOR.xyz = col;
}
```

```
shader_type canvas_item;

uniform sampler2D screen_texture : hint_screen_texture,
repeat_disable, filter_nearest;

// Blurs the screen in the Y-direction.
void fragment() {
    vec3 col = texture(screen_texture, SCREEN_UV).xyz * 0.16;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0,
SCREEN_PIXEL_SIZE.y)).xyz * 0.15;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, -
SCREEN_PIXEL_SIZE.y)).xyz * 0.15;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 2.0 *
SCREEN_PIXEL_SIZE.y)).xyz * 0.12;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 2.0 *
-SCREEN_PIXEL_SIZE.y)).xyz * 0.12;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 3.0 *
SCREEN_PIXEL_SIZE.y)).xyz * 0.09;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 3.0 *
-SCREEN_PIXEL_SIZE.y)).xyz * 0.09;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 4.0 *
SCREEN_PIXEL_SIZE.y)).xyz * 0.05;
    col += texture(screen_texture, SCREEN_UV + vec2(0.0, 4.0 *
-SCREEN_PIXEL_SIZE.y)).xyz * 0.05;
    COLOR.xyz = col;
}
```

Using the above code, you should end up with a full screen blur effect like below.

Advanced post-processing

Introduction

This tutorial describes an advanced method for post-processing in Godot. In particular, it will explain how to write a post-processing shader that uses the depth buffer. You should already be familiar with post-processing generally and, in particular, with the methods outlined in the custom post-processing tutorial.

Full screen quad

One way to make custom post-processing effects is by using a viewport. However, there are two main drawbacks of using a Viewport:

1. The depth buffer cannot be accessed
2. The effect of the post-processing shader is not visible in the editor

To get around the limitation on using the depth buffer, use a MeshInstance3D with a QuadMesh primitive. This allows us to use a shader and to access the depth texture of the scene. Next, use a vertex shader to make the quad cover the screen at all times so that the post-processing effect will be applied at all times, including in the editor.

First, create a new MeshInstance3D and set its mesh to a QuadMesh. This creates a quad centered at position `(0, 0, 0)` with a width and height of `1`. Set the width and height to `2` and enable **Flip Faces**. Right now, the quad occupies a position in world space at the origin. However, we want it to move with the camera so that it always covers the entire screen. To do this, we will bypass the coordinate transforms that translate the vertex positions through the difference coordinate spaces and treat the vertices as if they were already in clip space.

The vertex shader expects coordinates to be output in clip space, which are coordinates ranging from `-1` at the left and bottom of the screen to `1` at the top and right of the screen. This is why the QuadMesh needs to have height and width of `2`. Godot handles the transform from model to view space to clip space behind the scenes, so we need to nullify the effects of Godot's transformations. We do this by setting the `POSITION` built-in to our desired position. `POSITION` bypasses the built-in transformations and sets the vertex position in clip space directly.

```
shader_type spatial;
// Prevent the quad from being affected by lighting and fog.
This also improves performance.
render_mode unshaded, fog_disabled;

void vertex() {
  POSITION = vec4(VERTEX.xy, 1.0, 1.0);
}
```

Note

In versions of Godot earlier than 4.3, this code recommended using `POSITION = vec4(VERTEX, 1.0);` which implicitly assumed the clip-space near plane was at `0.0`. That code is now incorrect and will not work in versions 4.3+ as we use a "reversed-z" depth buffer now where the near plane is at `1.0`.

Even with this vertex shader, the quad keeps disappearing. This is due to frustum culling, which is done on the CPU. Frustum culling uses the camera matrix and the AABBs of Meshes to determine if the Mesh will be visible *before* passing it to the GPU. The CPU has no knowledge of what we are doing with the vertices, so it assumes the coordinates specified refer to world positions, not clip space positions, which results in Godot culling the quad when we turn away from the center of the scene. In order to keep the quad from being culled, there are a few options:

1. Add the QuadMesh as a child to the camera, so the camera is always pointed at it
2. Set the Geometry property `extra_cull_margin` as large as possible in the QuadMesh

The second option ensures that the quad is visible in the editor, while the first option guarantees that it will still be visible even if the camera moves outside the cull margin. You can also use both options.

Depth texture

To read from the depth texture, we first need to create a texture uniform set to the depth buffer by using `hint_depth_texture`.

```
uniform sampler2D depth_texture : source_color,
hint_depth_texture;
```

Once defined, the depth texture can be read with the `texture()` function.

```
float depth = texture(depth_texture, SCREEN_UV).x;
```

Note

Similar to accessing the screen texture, accessing the depth texture is only possible when reading from the current viewport. The depth texture cannot be accessed from another viewport to which you have rendered.

The values returned by `depth_texture` are between `1.0` and `0.0` (corresponding to the near and far plane, respectively, because of using a "reverse-z" depth buffer) and are nonlinear. When displaying depth directly from the `depth_texture`, everything will look almost black unless it is very close due to that nonlinearity. In order to make the depth value align with world or model coordinates, we need to linearize the value. When we apply the projection matrix to the vertex position, the z value is made nonlinear, so to linearize it, we multiply it by the inverse of the projection matrix, which in Godot, is accessible with the variable `INV_PROJECTION_MATRIX`.

Firstly, take the screen space coordinates and transform them into normalized device coordinates (NDC). NDC run `-1.0` to `1.0` in `x` and `y` directions and from `0.0` to `1.0` in the `z` direction when using the Vulkan backend. Reconstruct the NDC using `SCREEN_UV` for the `x` and `y` axis, and the depth value for `z`.

Note

This tutorial assumes the use of the Vulkan renderer, which uses NDCs with a Z-range of `[0.0, 1.0]`. In contrast, OpenGL uses NDCs with a Z-range of `[-1.0, 1.0]`.

```
void fragment() {
  float depth = texture(depth_texture, SCREEN_UV).x;
  vec3 ndc = vec3(SCREEN_UV * 2.0 - 1.0, depth);
}
```

Convert NDC to view space by multiplying the NDC by `INV_PROJECTION_MATRIX`. Recall that view space gives positions relative to the camera, so the `z` value will give us the distance to the point.

```
void fragment() {
  ...
  vec4 view = INV_PROJECTION_MATRIX * vec4(ndc, 1.0);
  view.xyz /= view.w;
  float linear_depth = -view.z;
}
```

Because the camera is facing the negative `z` direction, the position will have a negative `z` value. In order to get a usable depth value, we have to negate `view.z`.

The world position can be constructed from the depth buffer using the following code, using the `INV_VIEW_MATRIX` to transform the position from view space into world space.

```
void fragment() {
  ...
  vec4 world = INV_VIEW_MATRIX * INV_PROJECTION_MATRIX *
vec4(ndc, 1.0);
  vec3 world_position = world.xyz / world.w;
}
```

Example shader

Once we add a line to output to `ALBEDO`, we have a complete shader that looks something like this. This shader lets you visualize the linear depth or world space coordinates, depending on which line is commented out.

```
shader_type spatial;
// Prevent the quad from being affected by lighting and fog.
This also improves performance.
render_mode unshaded, fog_disabled;

uniform sampler2D depth_texture : source_color,
hint_depth_texture;

void vertex() {
  POSITION = vec4(VERTEX.xy, 1.0, 1.0);
}
```

```
void fragment() {
  float depth = texture(depth_texture, SCREEN_UV).x;
  vec3 ndc = vec3(SCREEN_UV * 2.0 - 1.0, depth);
  vec4 view = INV_PROJECTION_MATRIX * vec4(ndc, 1.0);
  view.xyz /= view.w;
  float linear_depth = -view.z;

  vec4 world = INV_VIEW_MATRIX * INV_PROJECTION_MATRIX *
vec4(ndc, 1.0);
  vec3 world_position = world.xyz / world.w;

  // Visualize linear depth
  ALBEDO.rgb = vec3(fract(linear_depth));

  // Visualize world coordinates
  //ALBEDO.rgb = fract(world_position).xyz;
}
```

An optimization

You can benefit from using a single large triangle rather than using a full screen quad. The reason for this is explained here [https://michaldrobot.com/2014/04/01/gcn-execution-patterns-in-full-screen-passes]. However, the benefit is quite small and only beneficial when running especially complex fragment shaders.

Set the Mesh in the MeshInstance3D to an ArrayMesh. An ArrayMesh is a tool that allows you to easily construct a Mesh from Arrays for vertices, normals, colors, etc.

Now, attach a script to the MeshInstance3D and use the following code:

```
extends MeshInstance3D

func _ready():
  # Create a single triangle out of vertices:
  var verts = PackedVector3Array()
  verts.append(Vector3(-1.0, -1.0, 0.0))
  verts.append(Vector3(-1.0, 3.0, 0.0))
  verts.append(Vector3(3.0, -1.0, 0.0))

  # Create an array of arrays.
  # This could contain normals, colors, UVs, etc.
  var mesh_array = []
  mesh_array.resize(Mesh.ARRAY_MAX) #required size for
ArrayMesh Array
  mesh_array[Mesh.ARRAY_VERTEX] = verts #position of vertex
array in ArrayMesh Array

  # Create mesh from mesh_array:
  mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES,
mesh_array)
```

Note

The triangle is specified in normalized device coordinates. Recall, NDC run from `-1.0` to `1.0` in both the `x` and `y` directions. This makes the screen `2` units wide and `2` units tall. In order to cover the entire screen with a single triangle, use a triangle that is `4` units wide and `4` units tall, double its height and width.

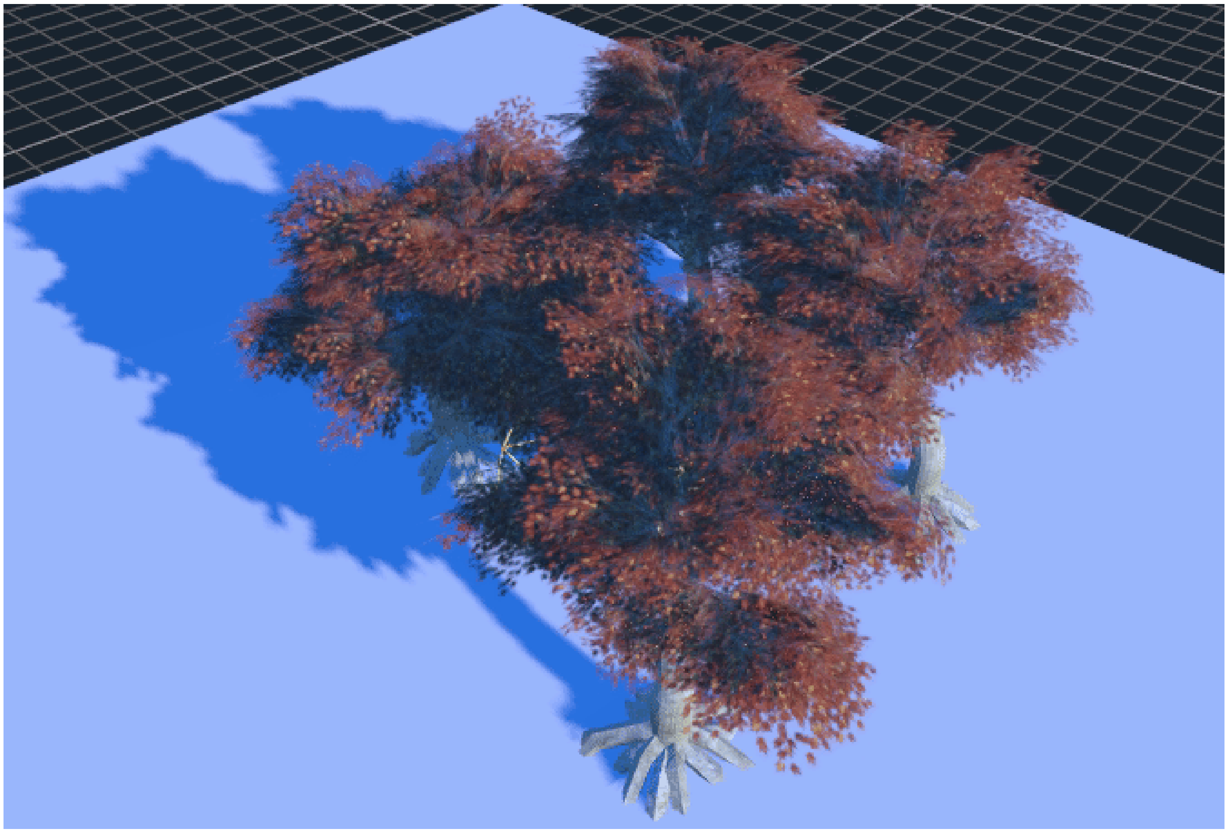
Assign the same vertex shader from above and everything should look exactly the same.

The one drawback to using an ArrayMesh over using a QuadMesh is that the ArrayMesh is not visible in the editor because the triangle is not constructed until the scene is run. To get around that, construct a single triangle Mesh in a modeling program and use that in the MeshInstance3D instead.

Making trees

This is a short tutorial on how to make trees and other types of vegetation from scratch.

The aim is to not focus on the modeling techniques (there are plenty of tutorials about that), but how to make them look good in Godot.



Start with a tree

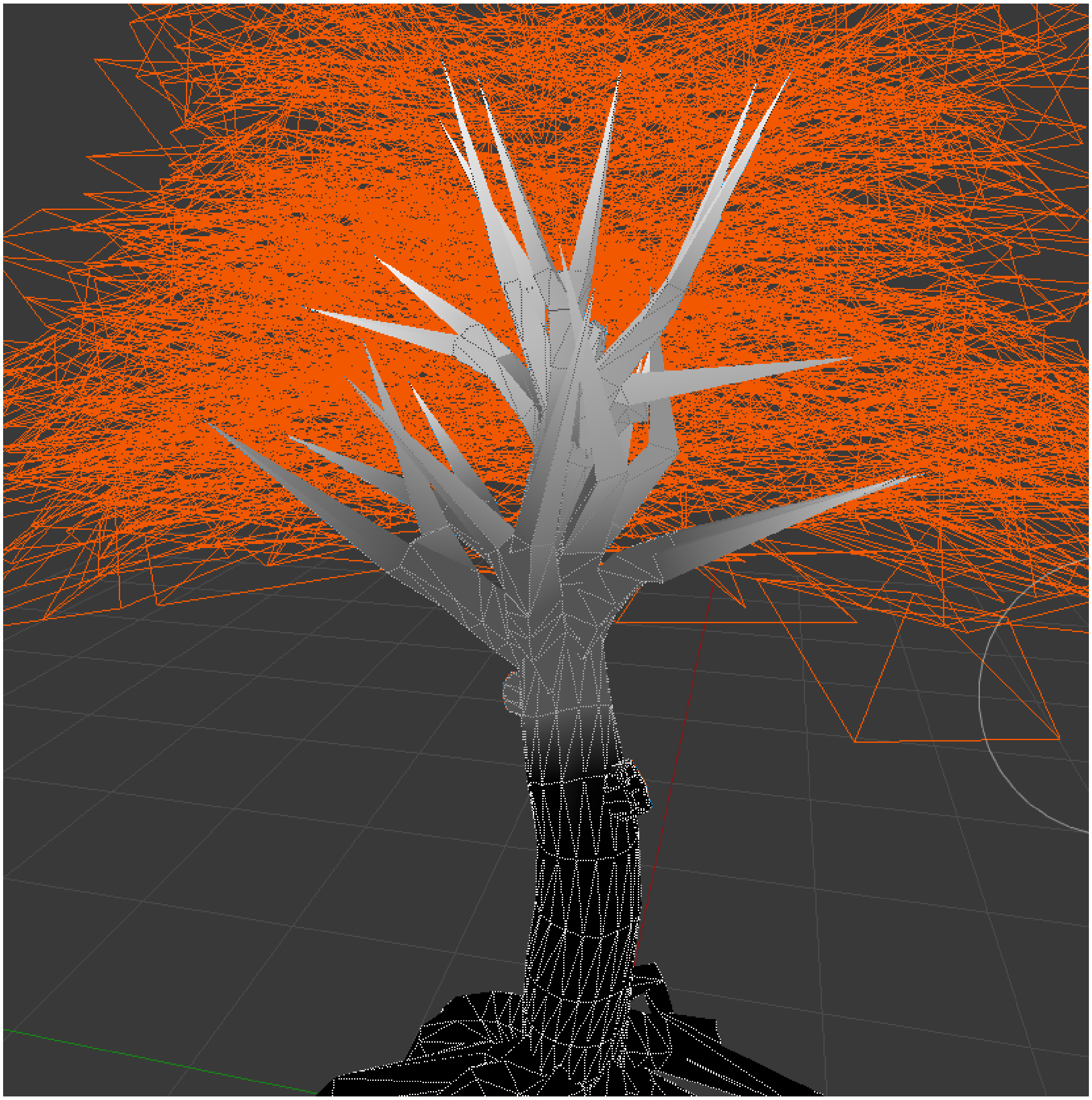
I took this tree from SketchFab:

https://sketchfab.com/models/ea5e6ed7f9d6445ba69589d503e8cebf

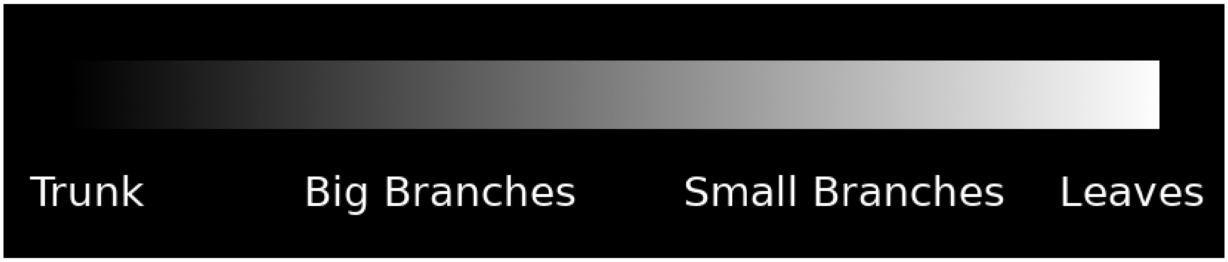
and opened it in Blender.

Paint with vertex colors

The first thing you may want to do is to use the vertex colors to paint how much the tree will sway when there is wind. Just use the vertex color painting tool of your favorite 3D modeling program and paint something like this:



This is a bit exaggerated, but the idea is that color indicates how much sway affects every part of the tree. This scale here represents it better:


Trunk
Big Branches
Small Branches
Leaves

Write a custom shader for the leaves

This is an example of a shader for leaves:

```
shader_type spatial;
render_mode depth_prepass_alpha, cull_disabled,
world_vertex_coords;
```

This is a spatial shader. There is no front/back culling (so leaves can be seen from both sides), and alpha prepass is used, so there are less depth artifacts that result from using transparency (and leaves cast shadow). Finally, for the sway effect, world coordinates are recommended, so the tree can be duplicated, moved, etc. and it will still work together with other trees.

```
uniform sampler2D texture_albedo : source_color;
uniform vec4 transmission : source_color;
```

Here, the texture is read, as well as a transmission color, which is used to add some back-lighting to the leaves, simulating subsurface scattering.

```
uniform float sway_speed = 1.0;
uniform float sway_strength = 0.05;
uniform float sway_phase_len = 8.0;

void vertex() {
    float strength = COLOR.r * sway_strength;
    VERTEX.x += sin(VERTEX.x * sway_phase_len * 1.123 + TIME *
sway_speed) * strength;
    VERTEX.y += sin(VERTEX.y * sway_phase_len + TIME *
sway_speed * 1.12412) * strength;
    VERTEX.z += sin(VERTEX.z * sway_phase_len * 0.9123 + TIME *
sway_speed * 1.3123) * strength;
}
```

This is the code to create the sway of the leaves. It's basic (just uses a sinewave multiplying by the time and axis position, but works well). Notice that the strength is multiplied by the color. Every axis uses a different small near 1.0 multiplication factor so axes don't appear in sync.

Finally, all that's left is the fragment shader:

```
void fragment() {
    vec4 albedo_tex = texture(texture_albedo, UV);
    ALBEDO = albedo_tex.rgb;
    ALPHA = albedo_tex.a;
    METALLIC = 0.0;
    ROUGHNESS = 1.0;
    TRANSMISSION = transmission.rgb;
}
```

And this is pretty much it.

The trunk shader is similar, except it does not write to the alpha channel (thus no alpha prepass is needed) and does not require transmission to work. Both shaders can be improved by adding normal mapping, AO and other maps.

Improving the shader

There are many more resources on how to do this that you can read. Now that you know the basics, a recommended read is the chapter from GPU Gems3 about how Crysis does this (focus mostly on the sway code, as many other techniques shown there are obsolete):

https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch16.html

User interface (UI)

In this section of the tutorial we explain the basics of creating a graphical user interface (GUI) in Godot.

UI building blocks

Like everything else in Godot the user interface is built using nodes, specifically Control nodes. There are many different types of controls which are useful for creating specific types of GUIs. For simplicity we can separate them into two groups: content and layout.

Typical content controls include:

- Buttons
- Labels
- LineEdits and TextEdits

Typical layout controls include:

- BoxContainers
- MarginContainers
- ScrollContainers
- TabContainers
- Popups

The following pages explain the basics of using such controls.

- Size and anchors
- Using Containers
- Custom GUI controls
- Keyboard/Controller Navigation and Focus
- Control node gallery

GUI skinning and themes

Godot features an in-depth skinning/theming system for control nodes. The pages in this section explain the benefits of that system and how to set it up in your projects.

- Introduction to GUI skinning
- Using the theme editor
- Theme type variations
- Using Fonts

Control node tutorials

The following articles cover specific details of using particular control nodes.

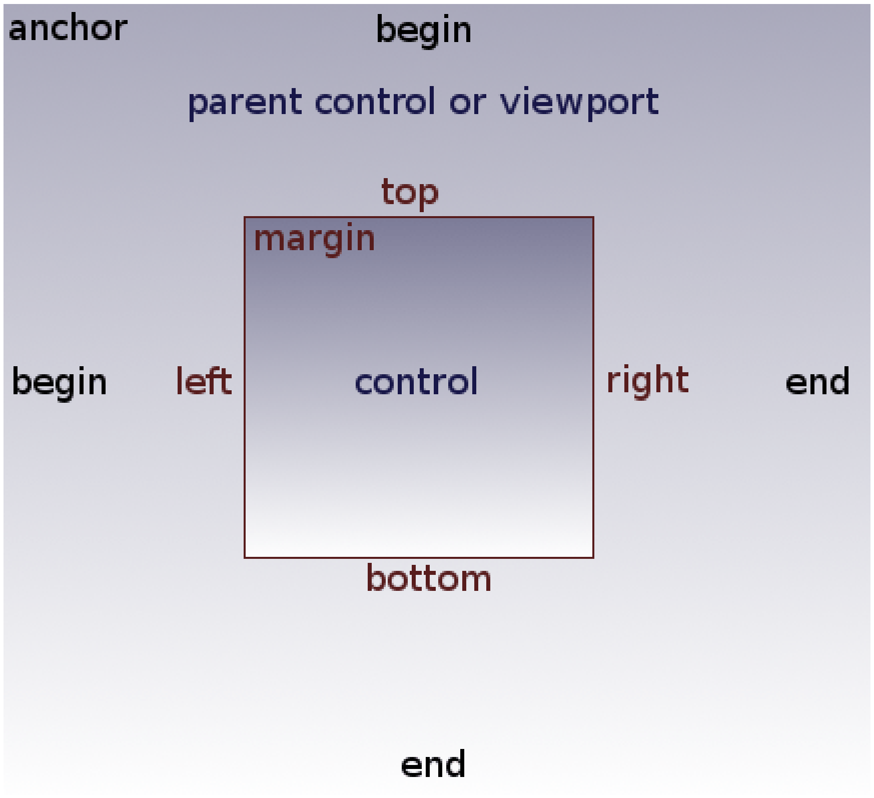
- BBCode in RichTextLabel

Size and anchors

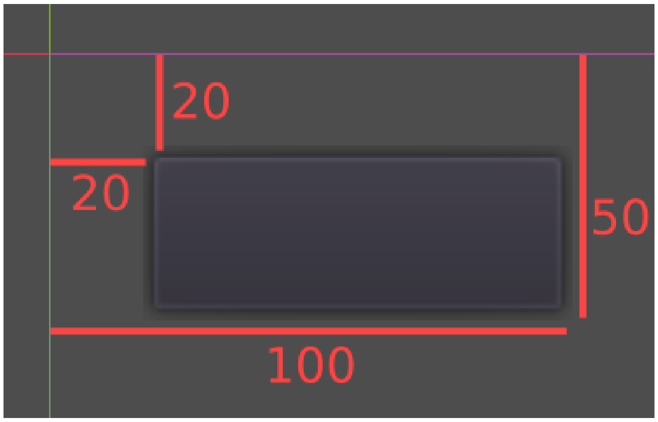
If a game was always going to be run on the same device and at the same resolution, positioning controls would be a simple matter of setting the position and size of each one of them. Unfortunately, that is rarely the case.

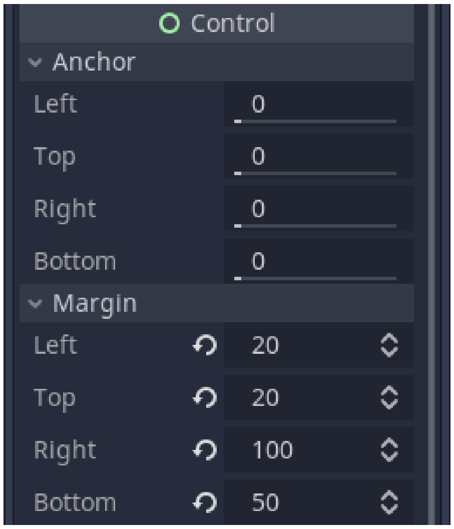
Only TVs nowadays have a standard resolution and aspect ratio. Everything else, from computer monitors to tablets, portable consoles and mobile phones have different resolutions and aspect ratios.

There are several ways to handle this, but for now, let's just imagine that the screen resolution has changed and the controls need to be re-positioned. Some will need to follow the bottom of the screen, others the top of the screen, or maybe the right or left margins.


anchor
begin
parent control or viewport
top
margin
begin
left
control
right
end
bottom
end

This is done by editing the *margin* properties of controls. Each control has four margins: left, right, bottom, and top, which correspond to the respective edges of the control. By default, all of them represent a distance in pixels relative to the top-left corner of the parent control or (in case there is no parent control) the viewport.




Control

Anchor
Left 0
Top 0
Right 0
Bottom 0

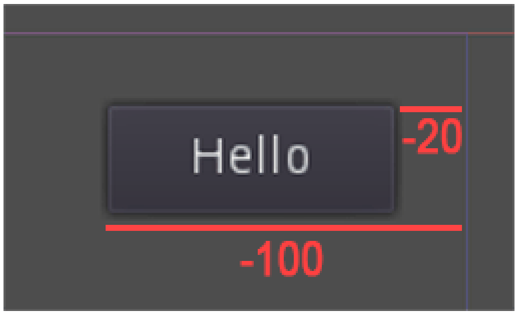
Margin
Left 20
Top 20
Right 100
Bottom 50

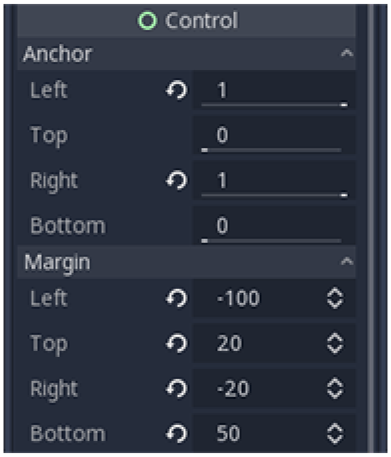
So to make the control wider you can make the right margin larger and/or make the left margin smaller. This lets you set the exact placement and shape of the control.

The *anchor* properties adjust where the margin distances are relative *to*. Each margin has an individual anchor that can be adjusted from the beginning to the end of the parent. So the vertical (top, bottom) anchors adjust from 0 (top of parent) to 1.0 (bottom of parent) with 0.5 being the center, and the control margins will be placed relative to that point. The horizontal (left, right) anchors similarly adjust from left to right of the parent.

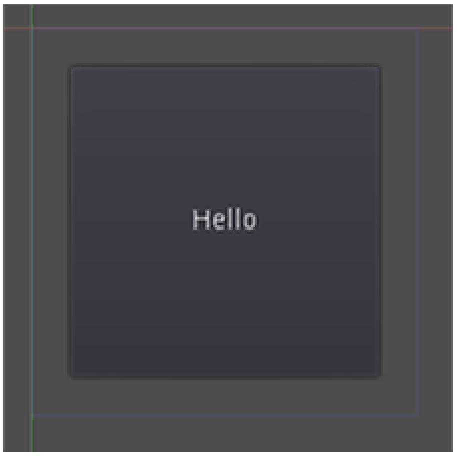
Note that when you wish the edge of a control to be above or left of the anchor point, you must change the margin value to be negative.

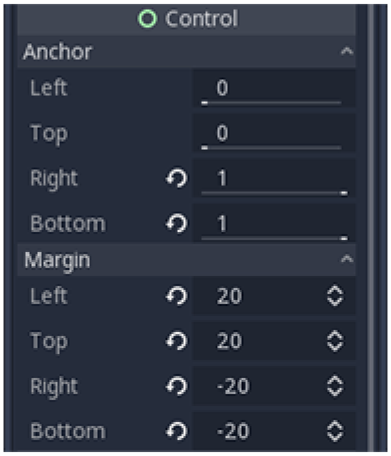
For example: when horizontal anchors are changed to 1, the margin values become relative to the top-right corner of the parent control or viewport.




Control
Anchor
Left 1
Top 0
Right 1
Bottom 0
Margin
Left -100
Top 20
Right -20
Bottom 50

Adjusting the two horizontal or the two vertical anchors to different values will make the control change size when the parent control does. Here, the control is set to anchor its bottom-right corner to the parent's bottom-right, while the top-left control margins are still anchored to the top-left of the parent, so when re-sizing the parent, the control will always cover it, leaving a 20 pixel margin:




Control
Anchor
Left 0
Top 0
Right 1
Bottom 1
Margin
Left 20
Top 20
Right -20
Bottom -20

Centering a control

To center a control in its parent, set its anchors to 0.5 and each margin to half of its relevant dimension. For example, the code below shows how a TextureRect can be centered in its parent:

GDScript

```
var rect = TextureRect.new()
rect.texture = load("res://icon.png")
rect.anchor_left = 0.5
rect.anchor_right = 0.5
rect.anchor_top = 0.5
rect.anchor_bottom = 0.5
var texture_size = rect.texture.get_size()
rect.offset_left = -texture_size.x / 2
rect.offset_right = texture_size.x / 2
rect.offset_top = -texture_size.y / 2
rect.offset_bottom = texture_size.y / 2
add_child(rect)
```

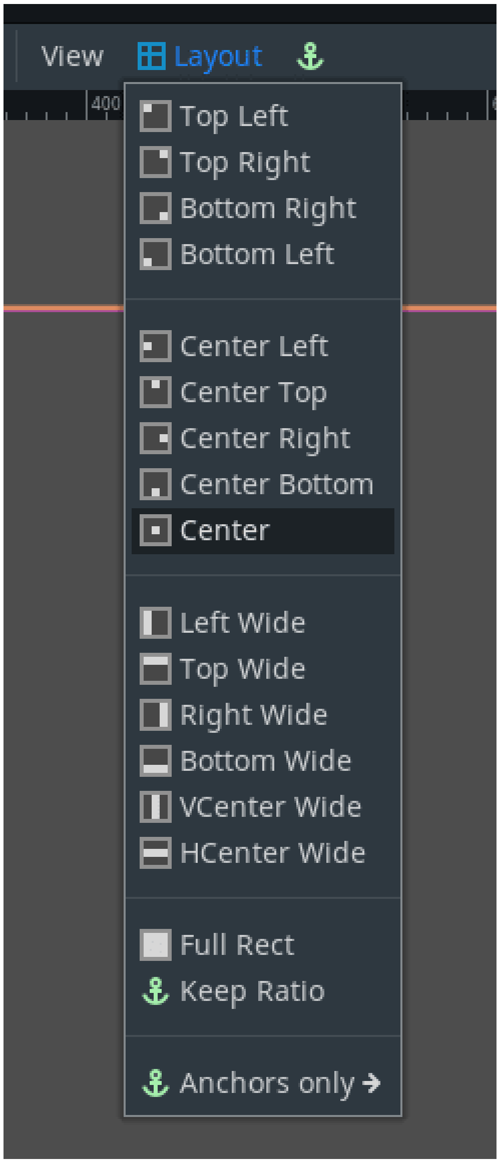
C#

```
var rect = new TextureRect();

rect.Texture = ResourceLoader.Load<Texture>("res://icon.png");
rect.AnchorLeft = 0.5f;
rect.AnchorRight = 0.5f;
rect.AnchorTop = 0.5f;
rect.AnchorBottom = 0.5f;

var textureSize = rect.Texture.GetSize();

rect.OffsetLeft = -textureSize.X / 2;
rect.OffsetRight = textureSize.X / 2;
rect.OffsetTop = -textureSize.Y / 2;
rect.OffsetBottom = textureSize.Y / 2;
AddChild(rect);
```

Setting each anchor to 0.5 moves the reference point for the margins to the center of its parent. From there, we set negative margins so that the control gets its natural size.

Layout Presets

Instead of manually adjusting the margin and anchor values, you can use the toolbar's Layout menu, above the viewport. Besides centering, it gives you many options to align and resize control nodes.


View
Layout
400
Top Left
Top Right
Bottom Right
Bottom Left
Center Left
Center Top
Center Right
Center Bottom
Center
Left Wide
Top Wide
Right Wide
Bottom Wide
VCenter Wide
HCenter Wide
Full Rect
Keep Ratio
Anchors only

Using Containers

Anchors are an efficient way to handle different aspect ratios for basic multiple resolution handling in GUIs.

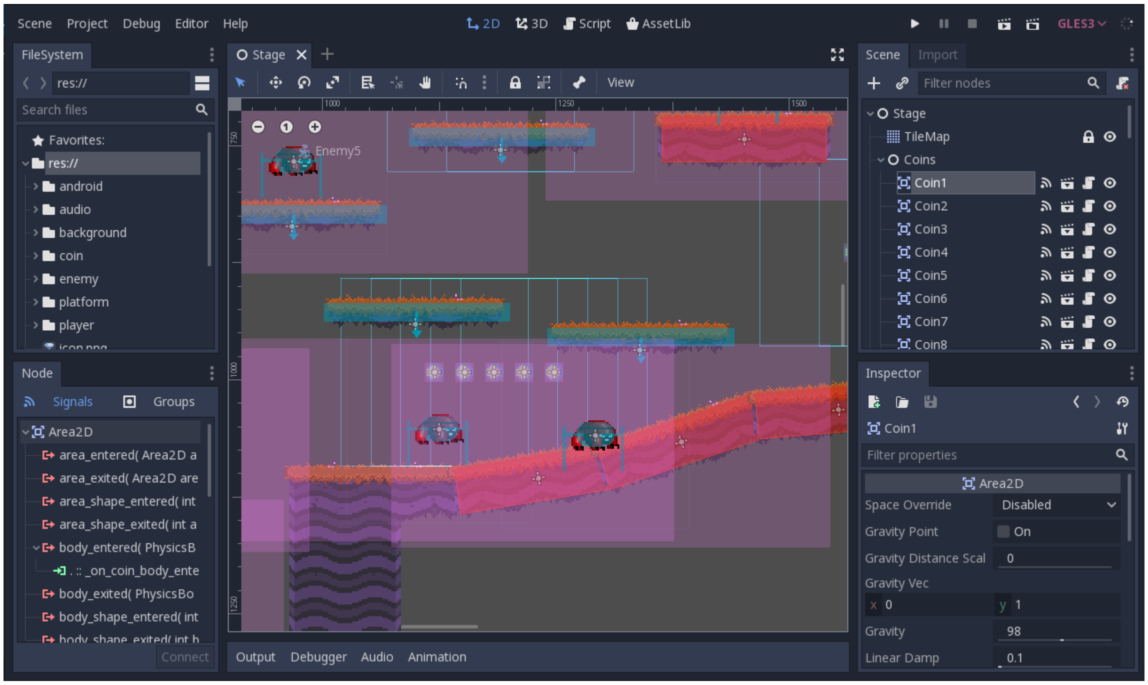
For more complex user interfaces, they can become difficult to use.

This is often the case of games, such as RPGs, online chats, tycoons or simulations. Another common case where more advanced layout features may be required is in-game tools (or simply just tools).

All these situations require a more capable OS-like user interface, with advanced layout and formatting. For that, Containers are more useful.

Container layout

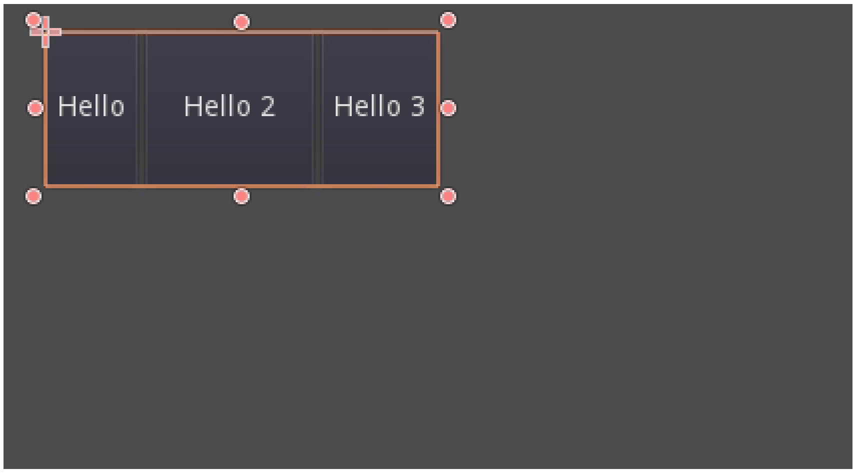
Containers provide a huge amount of layout power (as an example, the Godot editor user interface is entirely done using them):


Scene Project Debug Editor Help
2D 3D Script AssetLib
GLES3
FileSystem
res://
Search files
Favorites:
res://
android
audio
background
coin
enemy
platform
player
Stage
View
Enemy5
Scene Import
Filter nodes
Stage
TileMap
Coins
Coin1
Coin2
Coin3
Coin4
Coin5
Coin6
Coin7
Coin8
Node
Signals Groups
Area2D
area_entered(Area2D a
area_exited(Area2D are
area_shape_entered(int
area_shape_exited(int a
body_entered(PhysicsB
. :: _on_coin_body_ente
body_exited(PhysicsBo
body_shape_entered(int
body_shape_exited(int b
Connect
Output Debugger Audio Animation
Inspector
Coin1
Filter properties
Area2D

Space Override Disabled
Gravity Point On
Gravity Distance Scal 0
Gravity Vec x 0 y 1
Gravity 98
Linear Damp 0.1

When a Container-derived node is used, all children Control nodes give up their own positioning ability. This means the *Container* will control their positioning and any attempt to manually alter these nodes will be either ignored or invalidated the next time their parent is resized.

Likewise, when a *Container* derived node is resized, all its children will be re-positioned according to it, with a behavior based on the type of container used:

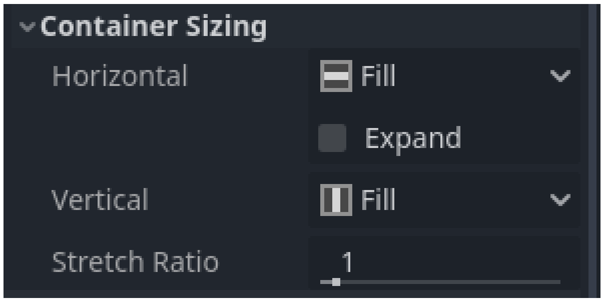

Hello
Hello 2
Hello 3

Example of *HBoxContainer* resizing children buttons.

The real strength of containers is that they can be nested (as nodes), allowing the creation of very complex layouts that resize effortlessly.

Sizing options

When adding a node to a container, the way the container treats each child depends mainly on their *container sizing options*. These options can be found by inspecting the layout of any *Control* that is a child of a *Container*.



Container Sizing
Horizontal: Fill
☐ Expand
Vertical: Fill
Stretch Ratio: 1

Sizing options are independent for vertical and horizontal sizing and not all containers make use of them (but most do):

- **Fill**: Ensures the control *fills* the designated area within the container. No matter if a control *expands* or not (see below), it will only *fill* the designated area when this is toggled on (it is by default).
- **Expand**: Attempts to use as much space as possible in the parent container (in each axis). Controls that don't expand will be pushed away by those that do. Between expanding controls, the amount of space they take from each other is determined by the *Stretch Ratio* (see below). This option is only available when the parent Container is of the right type, for example the *HBoxContainer* has this option for horizontal sizing.

- **Shrink Begin** When expanding, try to remain at the left or top of the expanded area.
- **Shrink Center** When expanding, try to remain at the center of the expanded area.
- **Shrink End** When expanding, try to remain at the right or bottom of the expanded area.
- **Stretch Ratio**: The ratio of how much expanded controls take up the available space in relation to each other. A control with "2", will take up twice as much available space as one with "1".

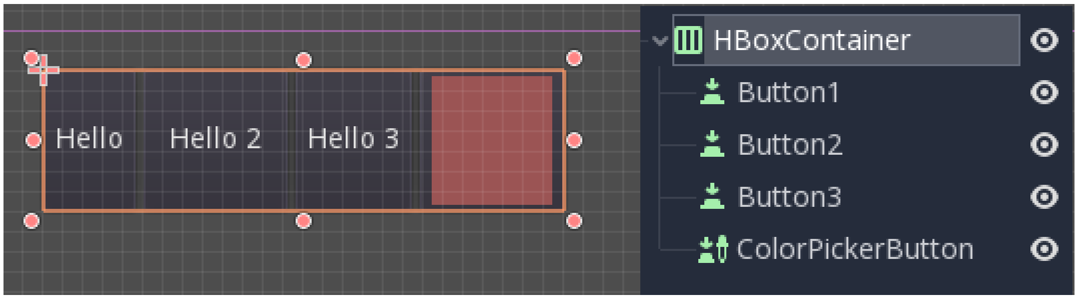
Experimenting with these flags and different containers is recommended to get a better grasp on how they work.

Container types

Godot provides several container types out of the box as they serve different purposes:

Box Containers

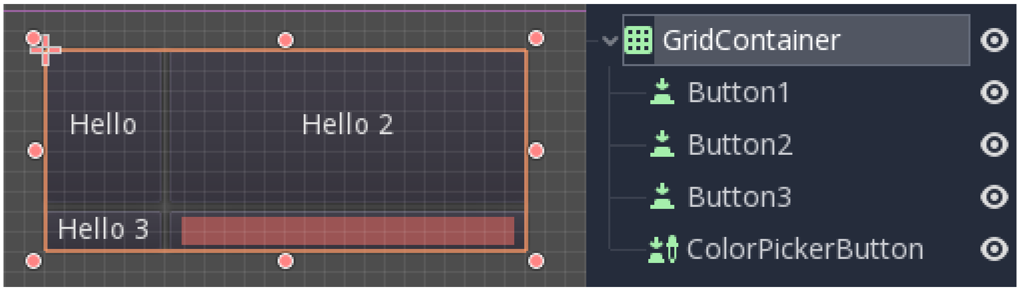
Arranges child controls vertically or horizontally (via HBoxContainer and VBoxContainer). In the opposite of the designated direction (as in, vertical for a horizontal container), it just expands the children.



These containers make use of the *Ratio* property for children with the *Expand* flag set.

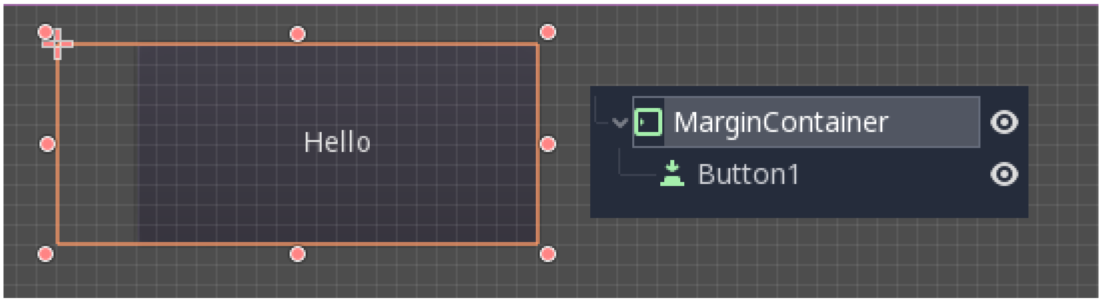
Grid Container

Arranges child controls in a grid layout (via GridContainer, amount of columns must be specified). Uses both the vertical and horizontal expand flags.

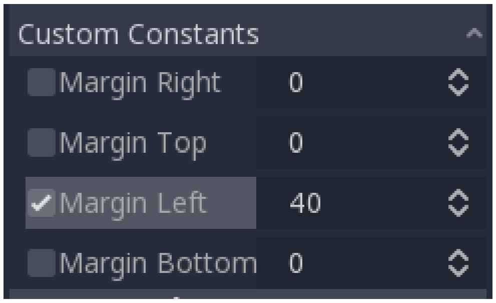

Hello
Hello 2
Hello 3
GridContainer
Button1
Button2
Button3
ColorPickerButton

Margin Container

Child controls are expanded towards the bounds of this control (via MarginContainer). Padding will be added on the margins depending on the theme configuration.


Hello
MarginContainer
Button1

Again, keep in mind that the margins are a *Theme* value, so they need to be edited from the constants overrides section of each control:


Custom Constants

☐ Margin Right 0

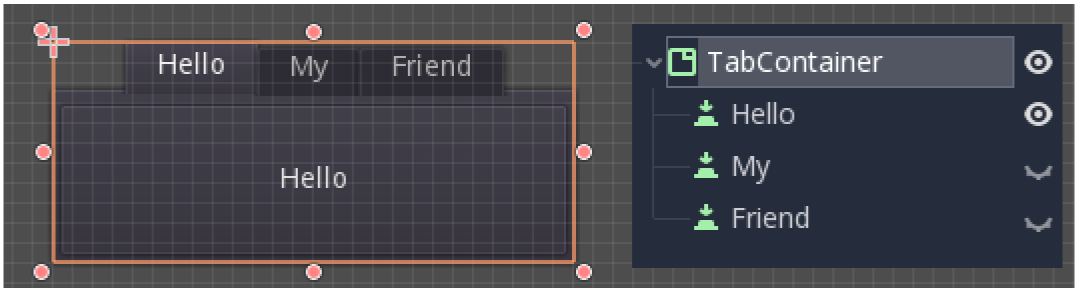
☐ Margin Top 0

☑ Margin Left 40

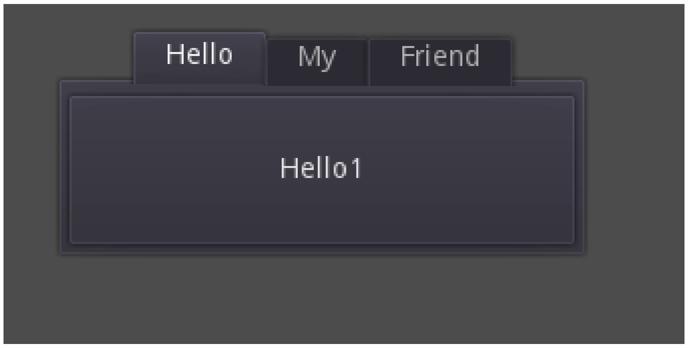
☐ Margin Bottom 0

Tab Container

Allows you to place several child controls stacked on top of each other (via TabContainer), with only the *current* one visible.


Hello
My
Friend
Hello
TabContainer
Hello
My
Friend

Changing the *current* one is done via tabs located at the top of the container, via clicking:

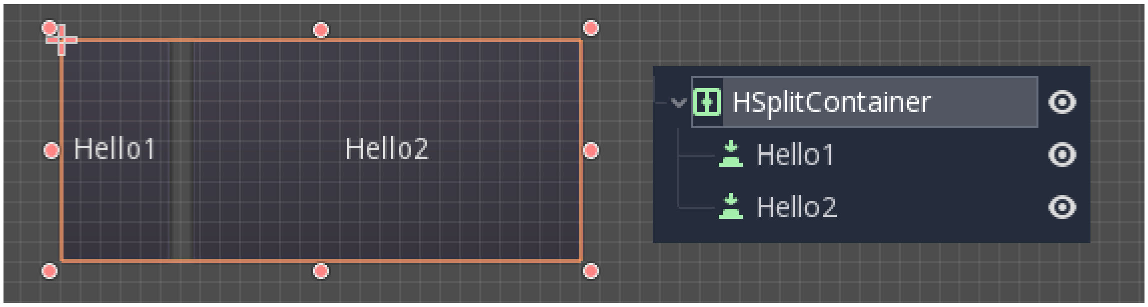

Hello
My
Friend
Hello1

The titles are generated from the node names by default (although they can be overridden via *TabContainer* API).

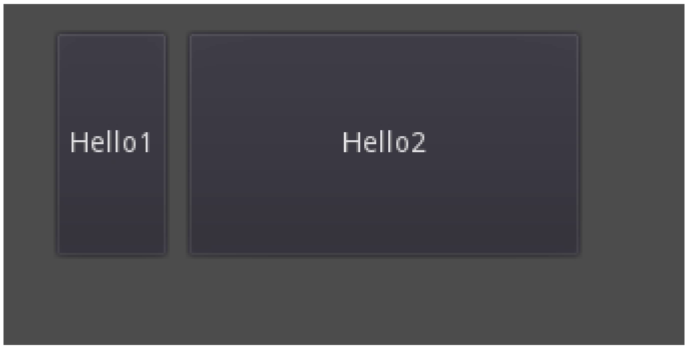
Settings such as tab placement and *StyleBox* can be modified in the *TabContainer* theme overrides.

Split Container

Accepts only one or two children controls, then places them side to side with a divisor (via HSplitContainer and VSplitContainer). Respects both horizontal and vertical flags, as well as *Ratio*.

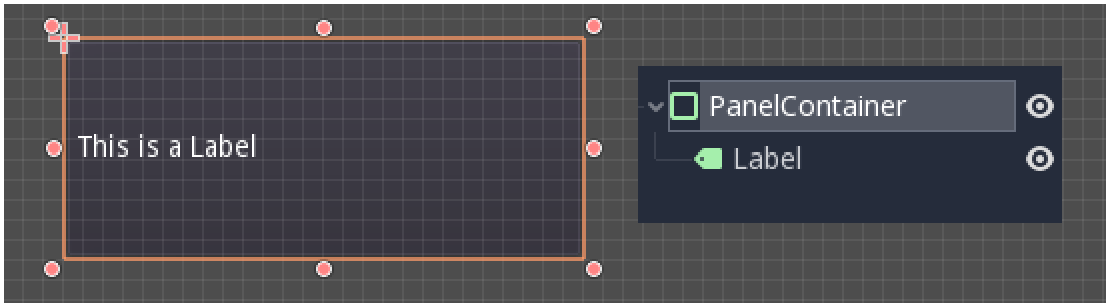

Hello1
Hello2
HSplitContainer
Hello1
Hello2

The divisor can be dragged around to change the size relation between both children:


Hello1
Hello2

PanelContainer

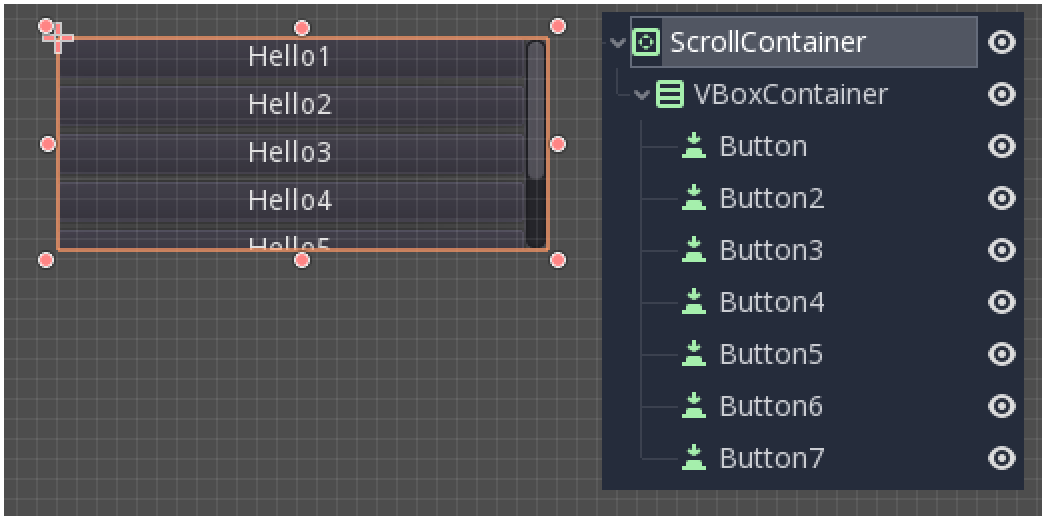
A container that draws a *StyleBox*, then expands children to cover its whole area (via PanelContainer, respecting the *StyleBox* margins). It respects both the horizontal and vertical sizing options.


This is a Label
PanelContainer
Label

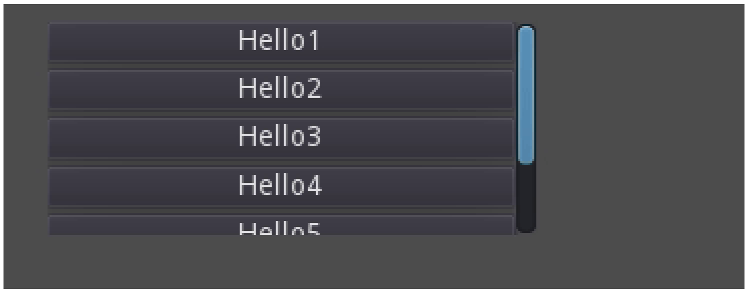
This container is useful as top-level, or just to add custom backgrounds to sections of a layout.

ScrollContainer

Accepts a single child node. If this node is bigger than the container, scrollbars will be added to allow panning the node around (via ScrollContainer). Both vertical and horizontal size options are respected, and the behavior can be turned on or off per axis in the properties.


Hello1
Hello2
Hello3
Hello4
Hello5
ScrollContainer
VBoxContainer
Button
Button2
Button3
Button4
Button5
Button6
Button7

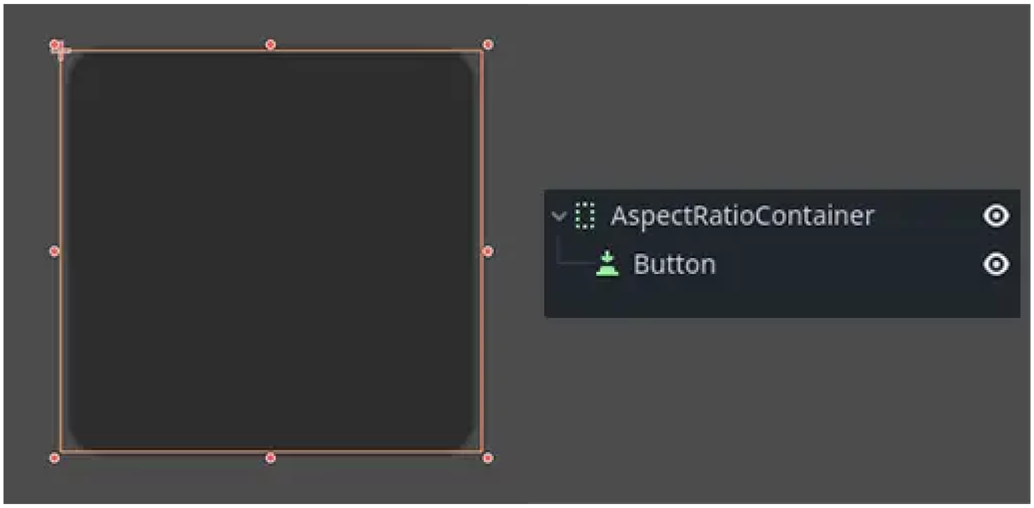
Mouse wheel and touch drag (when touch is available) are also valid ways to pan the child control around.



As in the example above, one of the most common ways to use this container is together with a *VBoxContainer* as child.

AspectRatioContainer

A container type that arranges its child controls in a way that preserves their proportions automatically when the container is resized. (via AspectRatioContainer). It has multiple stretch modes, providing options for adjusting the child controls' sizes concerning the container: "fill," "width control height," "height control width," and "cover."

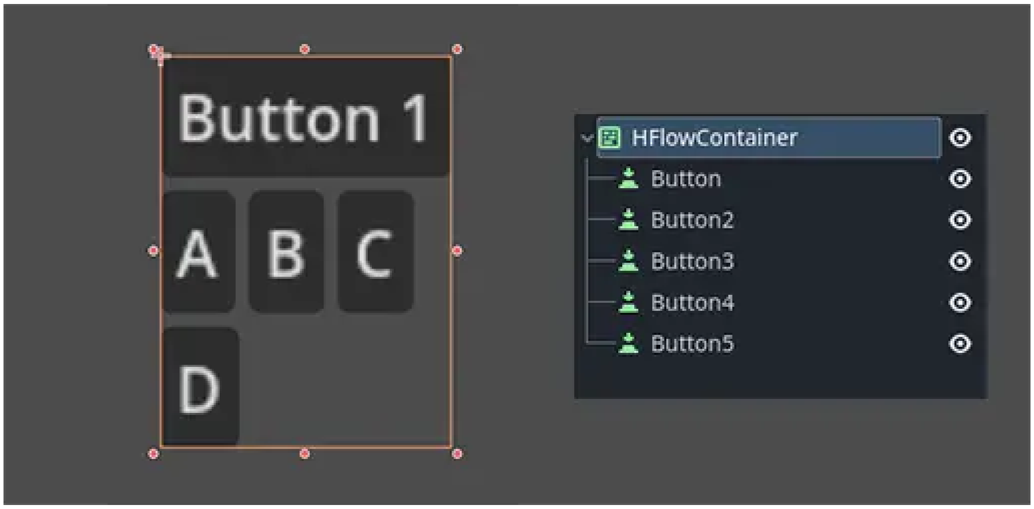


useful when you have a container that needs to be dynamic and responsive to different screen sizes, and you want the child elements to scale proportionally without losing their intended shapes.

tutorials/ui/img/containers_aspectratio_drag.png

FlowContainer

FlowContainer is a container that arranges its child controls either horizontally or vertically, (via HFlowContainer and via VFlowContainer). and when the available space runs out, it wraps the children to the next line or column, similar to how text wraps in a book.

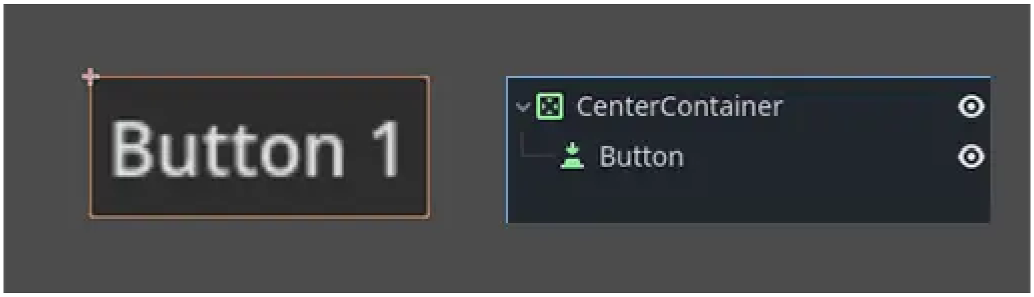


useful for creating flexible layouts where the child controls adjust automatically to the available space without overlapping.

tutorials/ui/img/containers_hflow_drag.png

CenterContainer

CenterContainer is a container that automatically keeps all of its child controls centered within it at their minimum size. It ensures that the child controls are always aligned to the center, making it easier to create centered layouts without manual positioning. (via CenterContainer).



tutorials/ui/img/containers_center_drag.png

SubViewportContainer

This is a special control that will only accept a single *Viewport* node as child, and it will display it as if it was an image (via SubViewportContainer).

Creating custom Containers

It is possible to create a custom container using a script. Here is an example of a container that fits children to its rect size:

GDScript

```
extends Container

func _notification(what):
    if what == NOTIFICATION_SORT_CHILDREN:
        # Must re-sort the children
        for c in get_children():
            # Fit to own size
            fit_child_in_rect(c, Rect2(Vector2(), rect_size))

func set_some_setting():
    # Some setting changed, ask for children re-sort.
    queue_sort()
```

C#

```csharp
using Godot;

public partial class CustomContainer : Container
{
    public override void _Notification(int what)
    {
        if (what == NotificationSortChildren)
        {
            // Must re-sort the children
            foreach (Control c in GetChildren())
            {
                // Fit to own size
                FitChildInRect(c, new Rect2(new Vector2(),
RectSize));
            }
        }
    }

    public void SetSomeSetting()
    {
        // Some setting changed, ask for children re-sort.
        QueueSort();
    }
}
```

Custom GUI controls

So many controls...

Yet there are never enough. Creating your own custom controls that act just the way you want them to is an obsession of almost every GUI programmer. Godot provides plenty of them, but they may not work exactly the way you want. Before contacting the developers with a pull-request to support diagonal scrollbars, at least it will be good to know how to create these controls easily from script.

Drawing

For drawing, it is recommended to check the Custom drawing in 2D tutorial. The same applies. Some functions are worth mentioning due to their usefulness when drawing, so they will be detailed next:

Checking control size

Unlike 2D nodes, "size" is important with controls, as it helps to organize them in proper layouts. For this, the Control.size property is provided. Checking it during `_draw()` is vital to ensure everything is kept in-bounds.

Checking focus

Some controls (such as buttons or text editors) might provide input focus for keyboard or joypad input. Examples of this are entering text or pressing a button. This is controlled with the Control.focus_mode property. When drawing, and if the control supports input focus, it is always desired to show some sort of indicator (highlight, box, etc.) to indicate that this is the currently focused control. To check for this status, the Control.has_focus() method exists. Example

GDScript

```
func _draw():
    if has_focus():
         draw_selected()
    else:
         draw_normal()
```

C#

```
public override void _Draw()
{
    if (HasFocus())
    {
        DrawSelected()
    }
    else
    {
        DrawNormal();
    }
}
```

Sizing

As mentioned before, size is important to controls. This allows them to lay out properly, when set into grids, containers, or anchored. Controls, most of the time, provide a *minimum size* to help properly lay them out. For example, if controls are placed vertically on top of each other using a VBoxContainer, the minimum size will make sure your custom control is not squished by the other controls in the container.

To provide this callback, just override Control._get_minimum_size(), for example:

GDScript

```
func _get_minimum_size():
    return Vector2(30, 30)
```

C#

```
public override Vector2 _GetMinimumSize()
{
    return new Vector2(20, 20);
}
```

Alternatively, set it using a function:

GDScript

```
func _ready():
    set_custom_minimum_size(Vector2(30, 30))
```

C#

```
public override void _Ready()
{
    CustomMinimumSize = new Vector2(20, 20);
}
```

Input

Controls provide a few helpers to make managing input events much easier than regular nodes.

Input events

There are a few tutorials about input before this one, but it's worth mentioning that controls have a special input method that only works when:

- The mouse pointer is over the control.

- The button was pressed over this control (control always captures input until button is released)
- Control provides keyboard/joypad focus via Control.focus_mode.

This function is Control._gui_input(). Simply override it in your control. No processing needs to be set.

GDScript

```
extends Control

func _gui_input(event):
   if event is InputEventMouseButton and event.button_index ==
MOUSE_BUTTON_LEFT and event.pressed:
       print("Left mouse button was pressed!")
```

C#

```
public override void _GuiInput(InputEvent @event)
{
    if (@event is InputEventMouseButton mbe && mbe.ButtonIndex
== MouseButton.Left && mbe.Pressed)
    {
        GD.Print("Left mouse button was pressed!");
    }
}
```

For more information about events themselves, check the Using InputEvent tutorial.

Notifications

Controls also have many useful notifications for which no dedicated callback exists, but which can be checked with the _notification callback:

GDScript

```
func _notification(what):
    match what:
        NOTIFICATION_MOUSE_ENTER:
            pass # Mouse entered the area of this control.
        NOTIFICATION_MOUSE_EXIT:
            pass # Mouse exited the area of this control.
        NOTIFICATION_FOCUS_ENTER:
            pass # Control gained focus.
        NOTIFICATION_FOCUS_EXIT:
            pass # Control lost focus.
        NOTIFICATION_THEME_CHANGED:
            pass # Theme used to draw the control changed;
            # update and redraw is recommended if using a
theme.
        NOTIFICATION_VISIBILITY_CHANGED:
            pass # Control became visible/invisible;
            # check new status with is_visible().
        NOTIFICATION_RESIZED:
            pass # Control changed size; check new size
            # with get_size().
        NOTIFICATION_MODAL_CLOSE:
            pass # For modal pop-ups, notification
            # that the pop-up was closed.
```

C#

```
public override void _Notification(int what)
{
    switch (what)
    {
        case NotificationMouseEnter:
            // Mouse entered the area of this control.
            break;

        case NotificationMouseExit:
            // Mouse exited the area of this control.
            break;

        case NotificationFocusEnter:
            // Control gained focus.
            break;

        case NotificationFocusExit:
            // Control lost focus.
            break;

        case NotificationThemeChanged:
            // Theme used to draw the control changed;
            // update and redraw is recommended if using a theme.
            break;

        case NotificationVisibilityChanged:
            // Control became visible/invisible;
            // check new status with is_visible().
            break;

        case NotificationResized:
            // Control changed size; check new size with get_size().
            break;

        case NotificationModalClose:
            // For modal pop-ups, notification that the pop-up was closed.
            break;
    }
}
```

Keyboard/Controller Navigation and Focus

It is a common requirement for a user interface to have full keyboard and controller support for navigation and interaction. There are two main reasons why this is beneficial for projects: improved accessibility (not everyone can use mouse or touch controls for interactions), and getting your project ready for consoles (or just for people who prefer to game with a controller on PC).

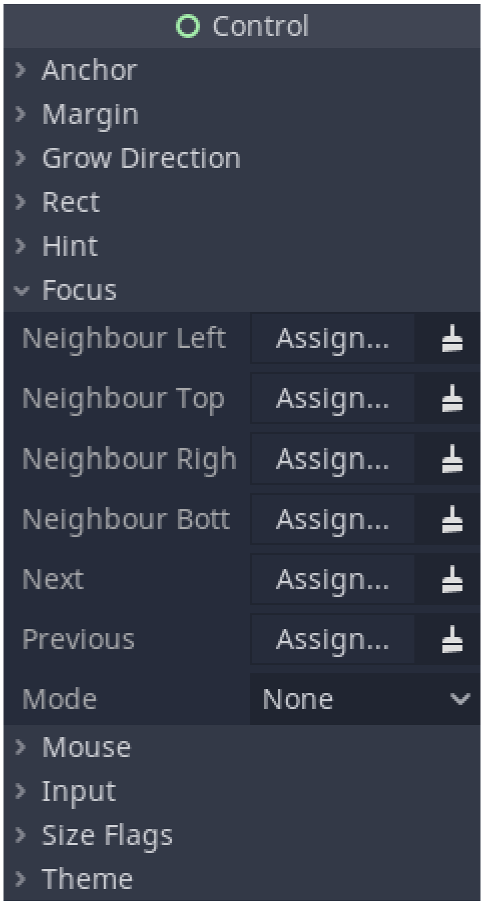
Navigating between UI elements with keyboard or controller is done by changing which node is actively selected. This is also called changing UI focus. Every Control node in Godot is capable of having focus. By default, some control nodes have the ability to automatically grab focus reacting to built-in UI actions such as `ui_up`, `ui_down`, `ui_focus_next`, etc. These actions can be seen in the project settings in the input map and can be modified.

Warning

Because these actions are used for focus they should not be used for any gameplay code.

Node settings

In addition to the built-in logic, you can define what is known as focus neighbors for each individual control node. This allows to finely tune the path the UI focus takes across the user interface of your project. The settings for individual nodes can be found in the Inspector dock, under the "Focus" category of the "Control" section.


Control
Anchor
Margin
Grow Direction
Rect
Hint
Focus
Neighbour Left
Assign...
Neighbour Top
Assign...
Neighbour Righ
Assign...
Neighbour Bott
Assign...
Next
Assign...
Previous
Assign...
Mode
None
Mouse
Input
Size Flags
Theme

Neighbor options are used to define nodes for 4-directional navigation, such as using arrow keys or a D-pad on a controller. For example, the bottom neighbor will be used when navigating down with the down arrow or by pushing down on the D-pad. The "Next" and "Previous" options are used with the focus shift button, such as `Tab` on desktop operating systems.

Note

A node can lose focus if it becomes hidden.

The mode setting defines how a node can be focused. **All** means a node can be focused by clicking on it with the mouse, or selecting it with a keyboard or controller. **Click** means it can only be focused on by clicking on it. Finally, **None** means it can't be focused at all.

Different control nodes have different default settings for this based on how they are typically used, for example, Label nodes are set to "None" by default, while buttons are set to "All".

Make sure to properly configure your scenes for focus and navigation. If a node has no focus neighbor configured, the engine will try to guess the next control automatically. This may result in unintended behavior, especially in a complex user interface that doesn't have well-defined vertical or horizontal navigation flow.

Necessary code

For keyboard and controller navigation to work correctly, any node must be focused on using code when the scene starts. Without doing this, pressing buttons or keys won't do anything. Here is a basic example of setting initial focus with code:

GDScript

```
func _ready():
    $StartButton.grab_focus()
```

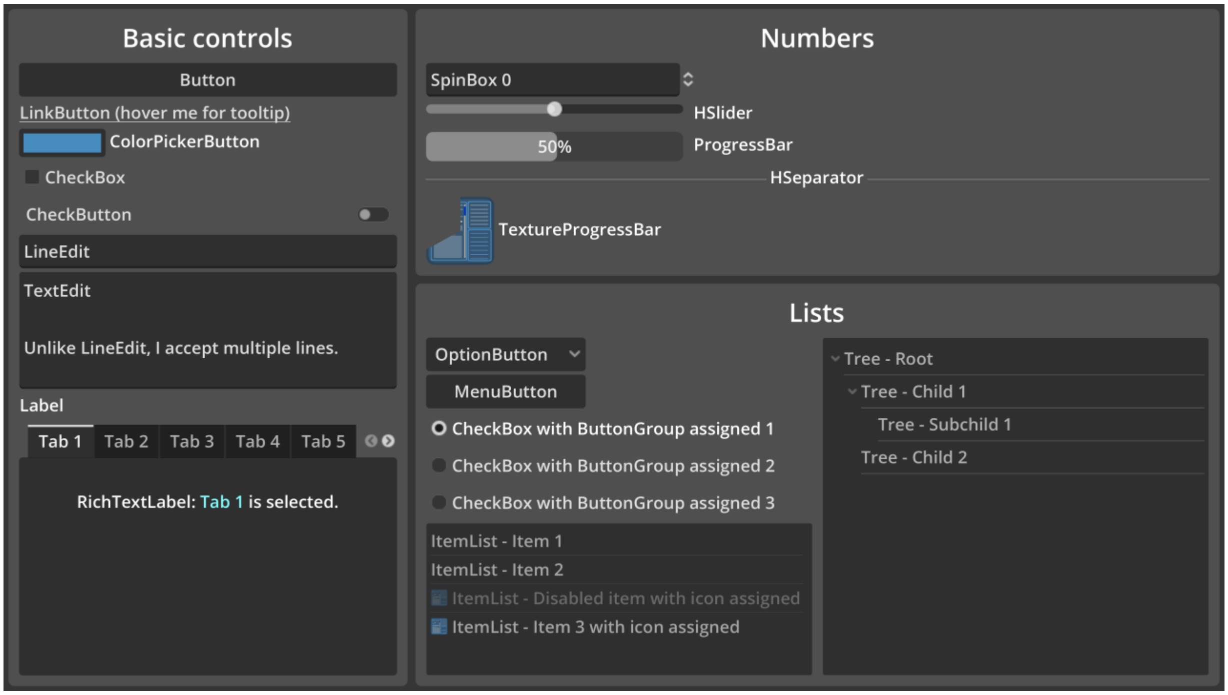
C#

```
public override void _Ready()
{
    GetNode<Button>("StartButton").GrabFocus();
}
```

Now when the scene starts the "Start Button" node will be focused, and the keyboard or a controller can be used to navigate between it and other UI elements.

Control node gallery

Here is a list of common Control nodes with their name next to them:


Basic controls
Button
LinkButton (hover me for tooltip)
ColorPickerButton
CheckBox
CheckButton
LineEdit
TextEdit
Label
Tab 1
Tab 2
Tab 3
Tab 4
Tab 5
Numbers
SpinBox 0
HSlider
50%
ProgressBar
HSeparator
TextureProgressBar
Lists
OptionButton
MenuButton
CheckBox with ButtonGroup assigned 1
CheckBox with ButtonGroup assigned 2
CheckBox with ButtonGroup assigned 3
ItemList - Item 1
ItemList - Item 2
ItemList - Disabled item with icon assigned
ItemList - Item 3 with icon assigned
Tree - Root
Tree - Child 1
Tree - Subchild 1
Tree - Child 2

Unlike LineEdit, I accept multiple lines.

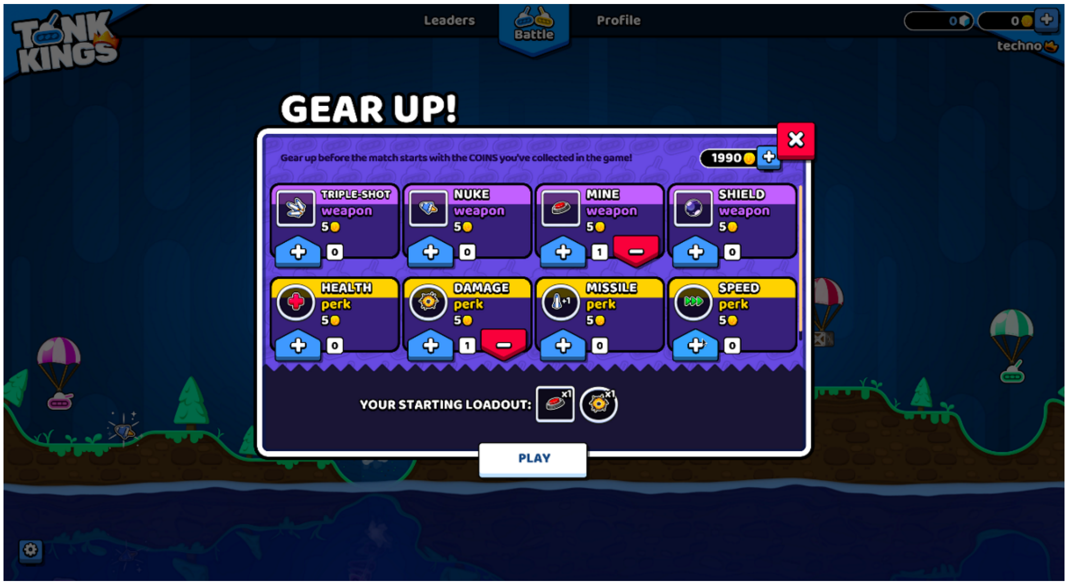
RichTextLabel: Tab 1 is selected.

The Control Gallery demo pictured above can be found on GitHub [https://github.com/godotengine/godot-demo-projects/tree/master/gui/control_gallery].

Introduction to GUI skinning

It is essential for a game to provide clear, informative, and yet visually pleasing user interface to its players. While Control nodes come with a decently functional look out of the box, there is always room for uniqueness and case-specific tuning. For this purpose Godot engine includes a system for GUI skinning (or theming), which allows you to customize the look of every control in your user interface, including your custom controls.

Here is an example of this system in action — a game with the GUI that is radically different from the default UI theme of the engine:


TANK KINGS
Leaders
Battle
Profile
0
0
techno
GEAR UP!
1990
TRIPLE-SHOT
weapon
5
0
NUKE
weapon
5
0
MINE
weapon
5
1
SHIELD
weapon
5
0
HEALTH
perk
5
0
DAMAGE
perk
5
1
MISSILE
perk
5
0
SPEED
perk
5
0
YOUR STARTING LOADOUT:
x1
x1
PLAY

Gear up before the match starts with the COINS you've collected in the game!

A "Gear Up!" screen in Tank Kings, courtesy of Winterpixel Games

Beyond achieving a unique look for your game, this system also enables developers to provide customization options to the end users, including accessibility settings. UI themes are applied in a cascading manner (i.e. they propagate from parent controls to their children), which means that font settings or adjustments for colorblind users can be applied in a single place and affect the entire UI tree. Of course this system can also be used for gameplay purposes: your hero-based game can change its style for the selected player character, or you can give different flavors to the sides in your team-based project.

Basics of themes

The skinning system is driven by the Theme resource. Every Godot project has an inherent default theme that contains the settings used by the built-in control nodes. This is what gives the controls their distinct look out of the box. A theme only describes the configuration, however, and it is still the job of each individual control to use that configuration in the way it requires to display itself. This is important to remember when implementing your own custom controls.

Note

Even the Godot editor itself relies on the default theme. But it doesn't look the same as a Godot project, because it applies its own heavily customized theme on top of the default one. In principle, this works exactly like it would in your game as explained below.

Theme items

The configuration that is stored in a theme consists of theme items. Each item has a unique name and must be one of the following data types:

- **Color**

A color value, which is often used for fonts and backgrounds. Colors can also be used for modulation of controls and icons.

- **Constant**

  An integer value, which can be used either for numeric properties of controls (such as the item separation in a BoxContainer), or for boolean flags (such as the drawing of relationship lines in a Tree).

- **Font**

  A font resource, which is used by controls that display text. Fonts contain most text rendering settings, except for its size and color. On top of that, alignment and text direction are controlled by individual controls.

- **Font size**

  An integer value, which is used alongside a font to determine the size at which the text should be displayed.

- **Icon**

  A texture resource, which is normally used to display an icon (on a Button, for example).

- **StyleBox**

  A StyleBox resource, a collection of configuration options which define the way a UI panel should be displayed. This is not limited to the Panel control, as styleboxes are used by many controls for their backgrounds and overlays.

Theme types

To help with the organization of its items each theme is separated into types, and each item must belong to a single type. In other words, each theme item is defined by its name, its data type and its theme type. This combination must be unique within the theme. For example, there cannot be two color items named `font_color` in a type called `Label`, but there can be another `font_color` item in a type `LineEdit`.

The default Godot theme comes with multiple theme types already defined, one for every built-in control node that uses UI skinning. The example above contains actual theme items present in the default theme. You can refer to the **Theme Properties** section in the class reference for each control to see which items are available to it and its child classes.

Note

Child classes can use theme items defined for their parent class (`Button` and its derivatives being a good example of that). In fact, every control can use every theme item of any theme type, if it needs to (but for the clarity and predictability we try to avoid that in the engine).

It is important to remember that for child classes that process is automated. Whenever a built-in control requests a theme item from the theme it can omit the theme type, and its class name will be used instead. On top of that, the class names of its parent classes will also be used in turn. This allows changes to the parent class, such as `Button`, to affect all derived classes without the need to customize every one of them.

You can also define your own theme types, and additionally customize both built-in controls and your own controls. Because built-in controls have no knowledge of your custom theme types, you must utilize scripts to access those items. All control nodes have several methods that allow to fetch theme items from the theme that is applied to them. Those methods accept the theme type as one of the arguments.

GDScript

```
var accent_color = get_theme_color("accent_color", "MyType")
label.add_color_override("font_color", accent_color)
```

C#

```
Color accentColor = GetThemeColor("accent_color", "MyType");
label.AddColorOverride("font_color", accentColor);
```

To give more customization opportunities types can also be linked together as type variations. This is another use-case for custom theme types. For example, a theme can contain a type `Header` which can be marked as a variation of the base `Label` type. An individual `Label` control can then be set to use the `Header` variation for its type, and every time a theme item is requested from a theme this variation will be used before any other type. This allows to store various presets of theme items for the same class of the control node in the single `Theme` resource.

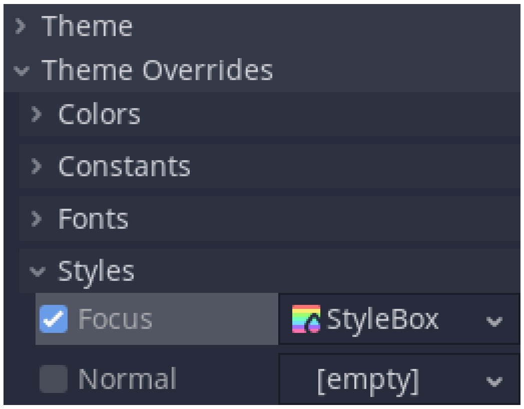
Warning

Only variations available from the default theme or defined in the custom project theme are shown in the Inspector dock as options. You can still input manually the name of a variation that is defined outside of those two places, but it is recommended to keep all variations to the project theme.

You can learn more about creating and using theme type variations in a dedicated article.

Customizing a control

Each control node can be customized directly without the use of themes. This is called local overrides. Every theme property from the control's class reference can be overridden directly on the control itself, using either the Inspector dock, or scripts. This allows to make granular changes to a particular part of the UI, while not affecting anything else in the project, including this control's children.



☑ Focus StyleBox

☐ Normal [empty]

Local overrides are less useful for the visual flair of your user interface, especially if you aim for consistency. However, for layout nodes these are essential. Nodes such as BoxContainer and GridContainer use theme constants for defining separation between their children, and MarginContainer stores its customizable margins in its theme items.

Whenever a control has a local theme item override, this is the value that it uses. Values provided by the theme are ignored.

Customizing a project

Out of the box each project adopts the default project theme provided by Godot. The default theme itself is constant and cannot be changed, but its items can be overridden with a custom theme.

Custom themes can be applied in two ways: as a project setting, and as a node property throughout the tree of control nodes.

There are two project settings that can be adjusted to affect your entire project: gui/theme/custom allows you to set a custom project-wide theme, and gui/theme/custom_font does the same to the default fallback font. When a theme item is requested by a control node the custom project theme, if present, is checked first. Only if it doesn't have the item the default theme is checked.

This allows you to configure the default look of every Godot control with a single theme resource, but you can go more granular than that. Every control node also has a theme property, which allows you to set a custom theme for the branch of nodes starting with that control. This means that the control and all of its children, and their children in turn, would first check that custom theme resource before falling back on the project and the default themes.

Note

Instead of changing the project setting you can set the custom theme resource to the root-most control node of your entire UI branch to almost the same effect. While in the running project it will behave as expected, individual scenes will still display using the default theme when previewing or running them directly. To fix that you can set the same theme resource to the root control of each individual scene.

For example, you can have a certain style for buttons in your project theme, but want a different look for buttons inside of a popup dialog. You can set a custom theme resource to the root control of your popup and define a different style for buttons within that resource. As long as the chain of control nodes between the root of the popup and the buttons is uninterrupted, those buttons will use the styles defined in the theme resource that is closest to them. All other controls will still be styled using the project-wide theme and the default theme styles.

To sum it up, for an arbitrary control its theme item lookup would look something like this:

1. Check for local overrides of the same data type and name.
2. Using control's type variation, class name and parent class names:
   a. Check every control starting from itself and see if it has a theme property set;
   b. If it does, check that theme for the matching item of the same name, data and theme type;
   c. If there is no custom theme or it doesn't have the item, move to the parent control;
   d. Repeat steps a-c. until the root of the tree is reached, or a non-control node is reached.
3. Using control's type variation, class name and parent class names check the project-wide theme, if it's present.
4. Using control's type variation, class name and parent class names check the default theme.

Even if the item doesn't exist in any theme, a corresponding default value for that data type will be returned.

Beyond controls

Naturally, themes are an ideal type of resource for storing configuration for something visual. While the support for theming is built into control nodes, other nodes can use them as well, just like any other resource.

An example of using themes for something beyond controls can be a modulation of sprites for the same units on different teams in a strategy game. A theme resource can define a collection of colors, and sprites (with a help from scripts) can use those colors to draw the texture. The main benefit being that you could make different themes using the same theme items for red, blue, and green teams, and swap them with a single resource change.

Using the theme editor

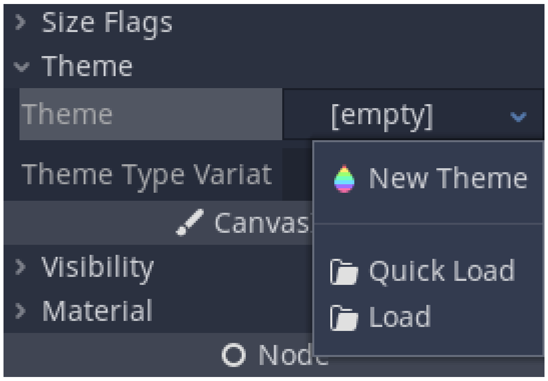
This article explains how to create and manage UI themes using the Godot editor and its theme editor tool. We recommend getting familiar with the basics behind GUI skinning/theming by reading Introduction to GUI skinning before starting.

The theme editor is a bottom panel tool that activates automatically, when a Theme resource is selected for editing. It contains the necessary UI for adding, removing, and adjusting theme types and theme items. It features a preview section for testing your changes live, as well as a window dialog for doing bulk operations of the theme items.

Creating a theme

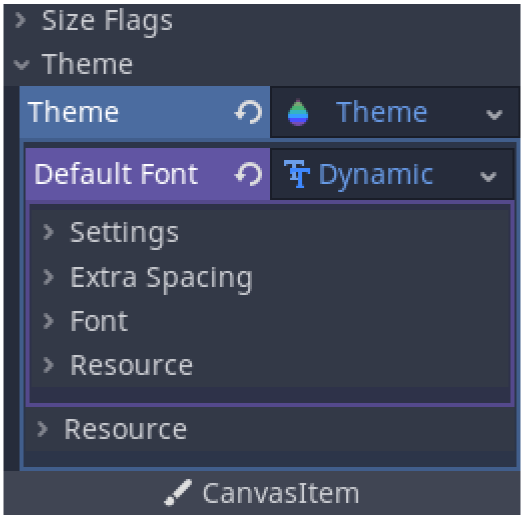
Like any other resources, themes can be created directly in the file system dock by right-clicking and selecting **New Resource...**, then selecting **Theme** and clicking **Create**. This is especially useful for creating project-wide themes.

Themes also can be created from any control node. Select a control node in the scene hierarchy, then in the inspector go to the `theme` property. From there you can select **New Theme**.

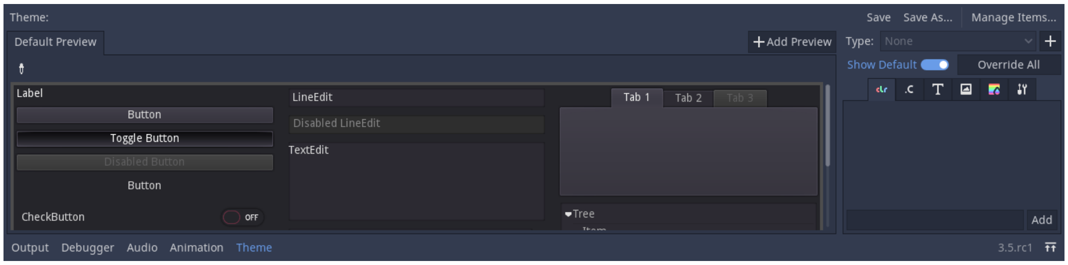


This will create an empty theme and open up the theme editor. Keep in mind that resources created this way are bundled with the scene by default. Use the context menu to save the new theme to a file instead.

While the theme editor provides the tools to manage theme types and items, themes also include the default, fallback font that you can edit only using the Inspector dock. Same applies to the contents of complex resource types, such as StyleBoxes and icons — they open for editing in the Inspector.


Size Flags
Theme
Theme
Theme
Default Font
Dynamic
Settings
Extra Spacing
Font
Resource
Resource
CanvasItem

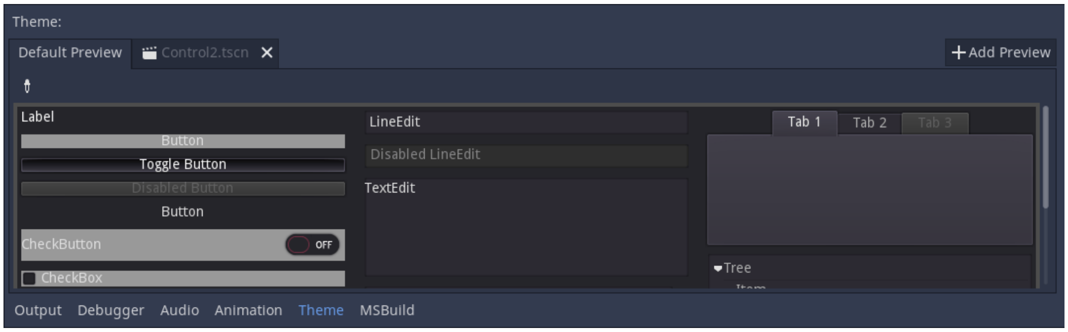
Theme editor overview


Theme:
Save
Save As...
Manage Items...
Default Preview
Add Preview
Type:
None
Show Default
Override All
Label
Button
Toggle Button
Disabled Button
Button
CheckButton
OFF
LineEdit
Disabled LineEdit
TextEdit
Tab 1
Tab 2
Tab 3
Tree
Add
Output
Debugger
Audio
Animation
Theme
3.5.rc1

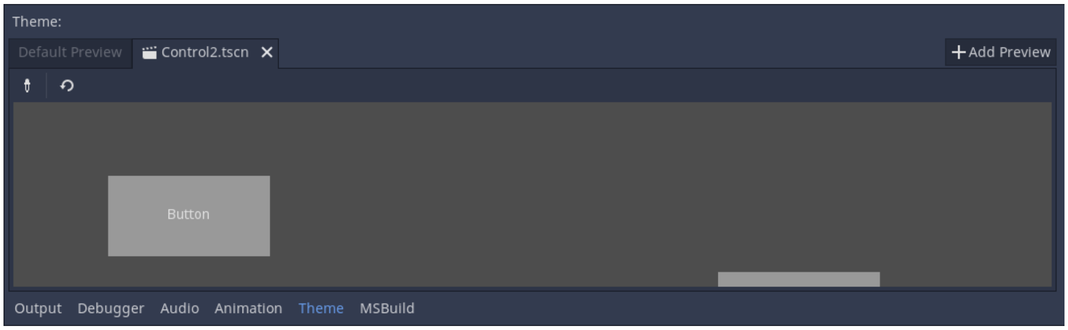
The theme editor has two main parts. The main theme editor, located at the bottom of the Godot editor, aims to provide users with tools to quickly create, edit, and delete theme items and types. It gives visual tools for picking and changing controls, abstracting the underlying theme concepts. The **Manage Theme Items** dialog, on the other hand, tries to address the needs of those who want to change themes manually. It's also useful for creating a new editor theme.

Theme previews

The left-hand side of the main editor has a set of preview tabs. The **Default Preview** tab is visible out of the box and contains most of the frequently used controls in various states. Previews are interactive, so intermediate states (e.g. hover) can be previewed as well.

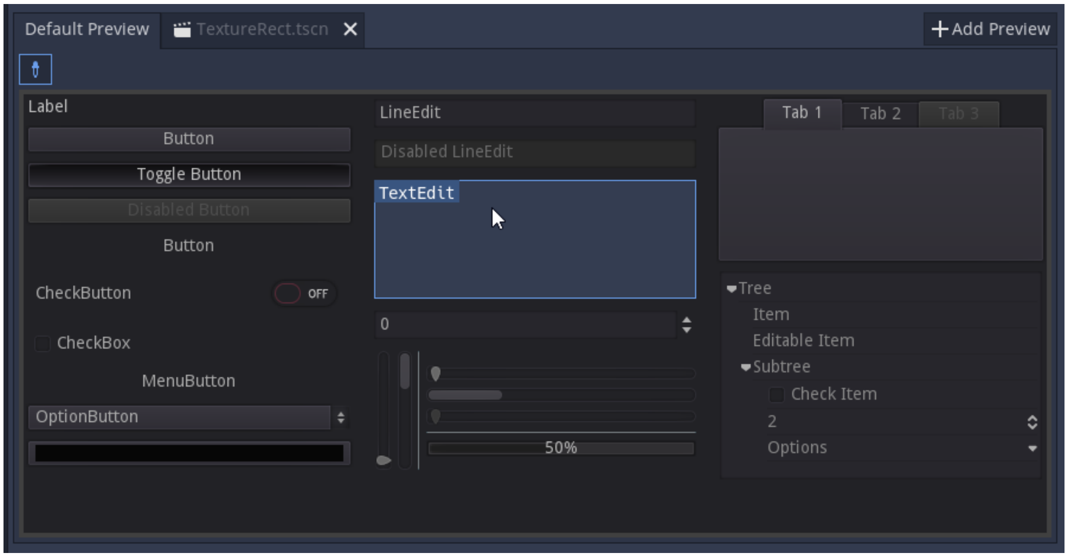

Theme:
Default Preview
Control2.tscn
Add Preview
Label
Button
Toggle Button
Disabled Button
Button
CheckButton
OFF
CheckBox
LineEdit
Disabled LineEdit
TextEdit
Tab 1
Tab 2
Tab 3
Tree
Output
Debugger
Audio
Animation
Theme
MSBuild

Additional tabs can be created from arbitrary scenes in your project. The scene must have a control node as its root to function as a preview. To add a new tab click the **Add Preview** button and select the saved scene from your file system.


Theme:
Default Preview
Control2.tscn
Add Preview
Button
Output
Debugger
Audio
Animation
Theme
MSBuild

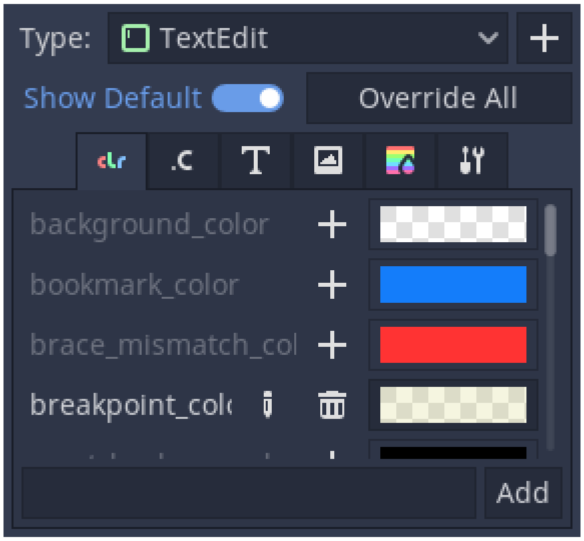
If you make changes to the scene, they will not be reflected in the preview automatically. To update the preview click the reload button on the toolbar.

Previews can also be used to quickly select the theme type to edit. Select the picker tool from the toolbar and hover over the preview area to highlight control nodes. Highlighted control nodes display their class name, or type variation if available. Clicking on the highlighted control opens it for editing on the right-hand side.


Default Preview
TextureRect.tscn
Add Preview
Label
Button
Toggle Button
Disabled Button
Button
CheckButton
OFF
CheckBox
MenuButton
OptionButton
LineEdit
Disabled LineEdit
TextEdit
0
50%
Tab 1
Tab 2
Tab 3
Tree
Item
Editable Item
Subtree
Check Item
2
Options

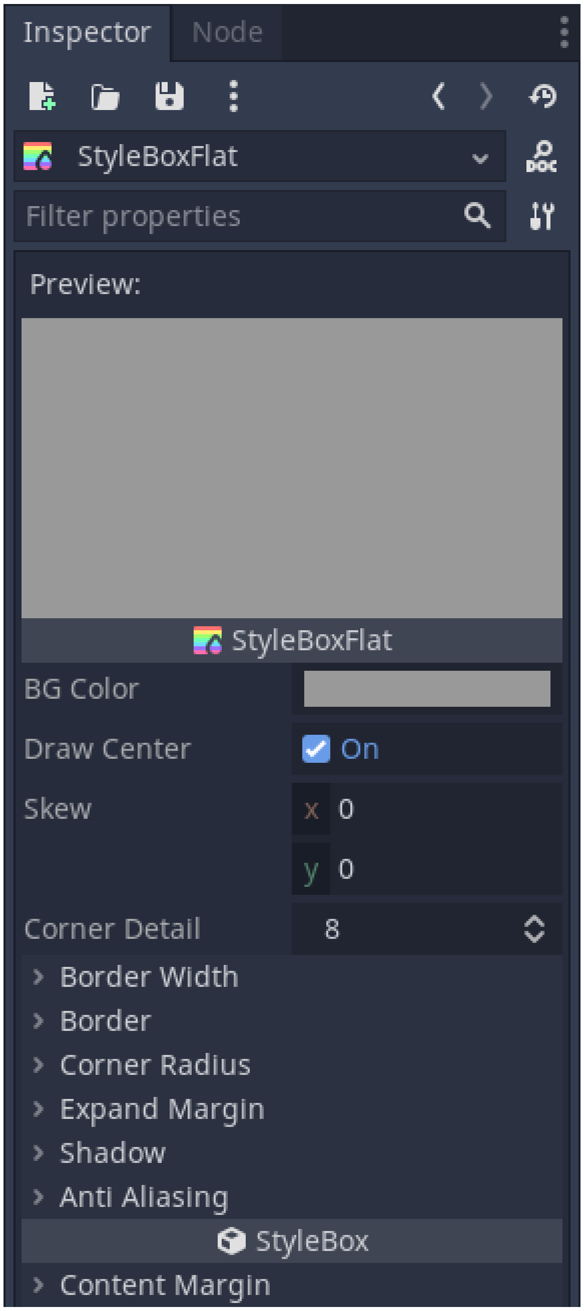
Theme types and items

The right-hand side of the theme editor provides a list of theme types available in the edited theme resource, and the contents of the selected type. The list of type's items is divided into several tabs, corresponding to each data type available in the theme (colors, constants, styles, etc.). If the **Show Default** option is enabled, then for each built-in type its default theme values are displayed, greyed out. If the option is disabled, only the items available in the edited theme itself are displayed.


Type:
TextEdit
Show Default
Override All
background_color
bookmark_color
brace_mismatch_col
breakpoint_colo
Add

Individual items from the default theme can be added to the current theme by clicking on the **Override** button next to the item. You can also override all the default items of the selected theme type by clicking on the **Override All** button. Overridden properties can then be removed with the **Remove Item** button. Properties can also be renamed using the **Rename Item** button, and completely custom properties can be added to the list using the text field below it.

Overridden theme items can be edited directly in the right-hand panel, unless they are resources. Resources have rudimentary controls available for them, but must be edited in the Inspector dock instead.


Inspector
Node
StyleBoxFlat
Filter properties
Preview:
StyleBoxFlat

BG Color

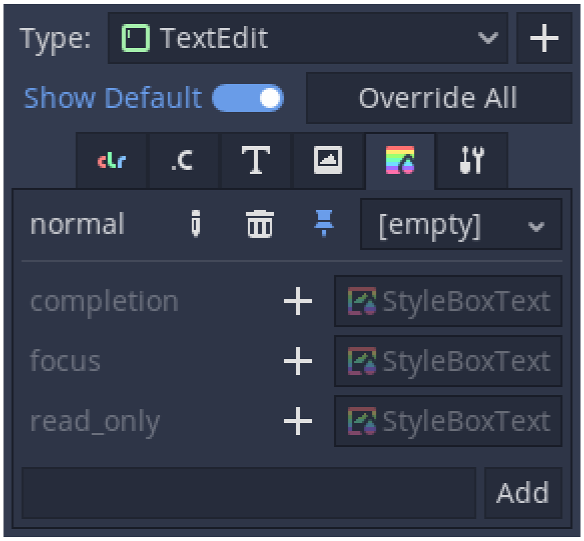
Draw Center ☑ On

Skew x 0 y 0

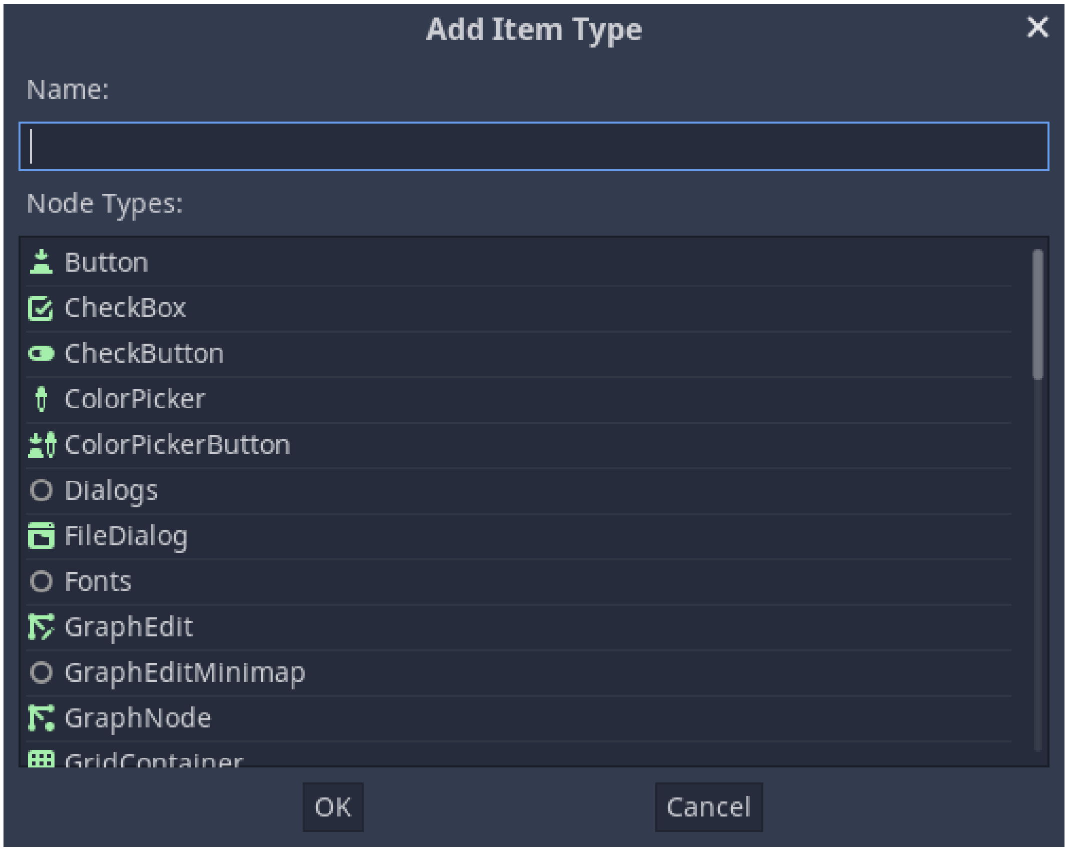
Corner Detail 8

Border Width
Border
Corner Radius
Expand Margin
Shadow
Anti Aliasing
StyleBox
Content Margin

Styleboxes have a unique feature available, where you can pin an individual stylebox from the list. Pinned stylebox acts like the leader of the pack, and all styleboxes of the same type are updated alongside it when you change its properties. This allows you to edit properties of several styleboxes at the same time.

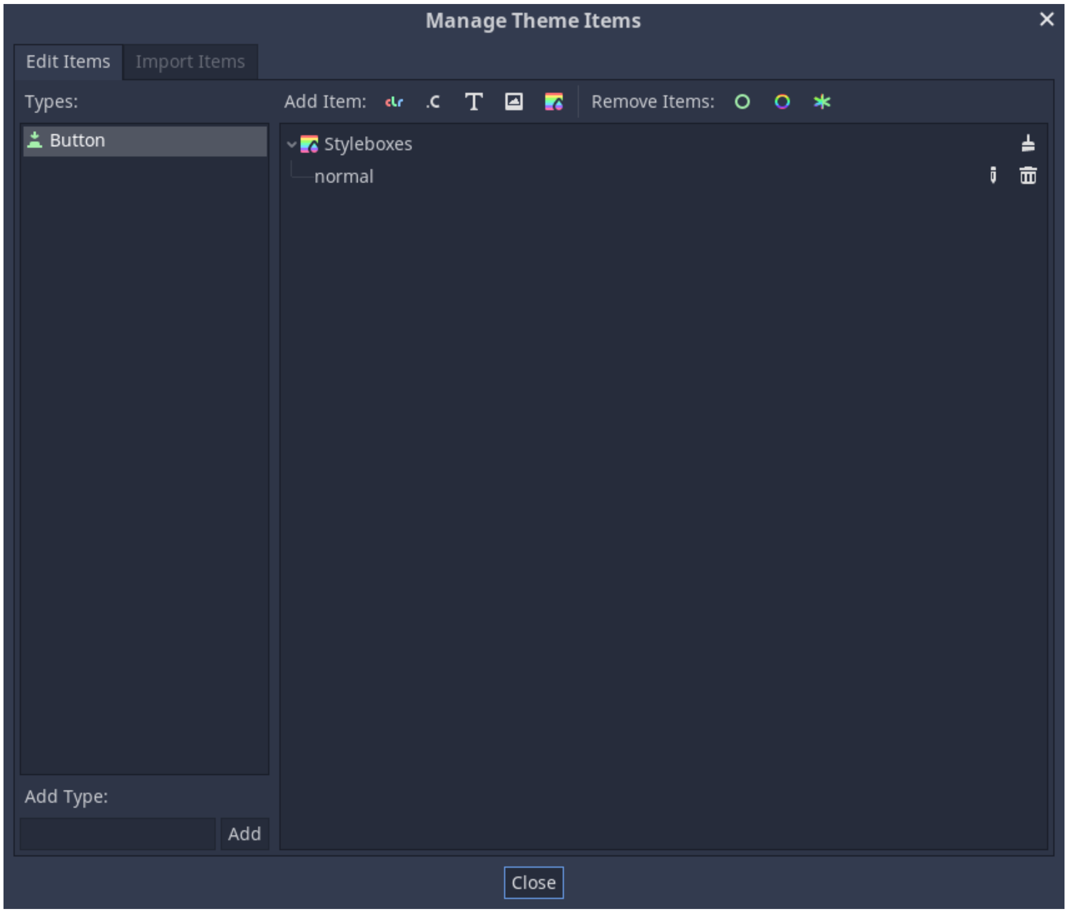

Type: TextEdit
Show Default
Override All
normal
[empty]
completion
StyleBoxText
focus
StyleBoxText
read_only
StyleBoxText
Add

While theme types can be picked from a preview, they can also be added manually. Clicking the plus button next to the type list opens the **Add item Type** menu. In that menu you can either select a type from the list, or you can enter an arbitrary name to create a custom type. Text field also filters the list of control nodes.


Add Item Type
Name:
Node Types:
Button
CheckBox
CheckButton
ColorPicker
ColorPickerButton
Dialogs
FileDialog
Fonts
GraphEdit
GraphEditMinimap
GraphNode
GridContainer
OK
Cancel

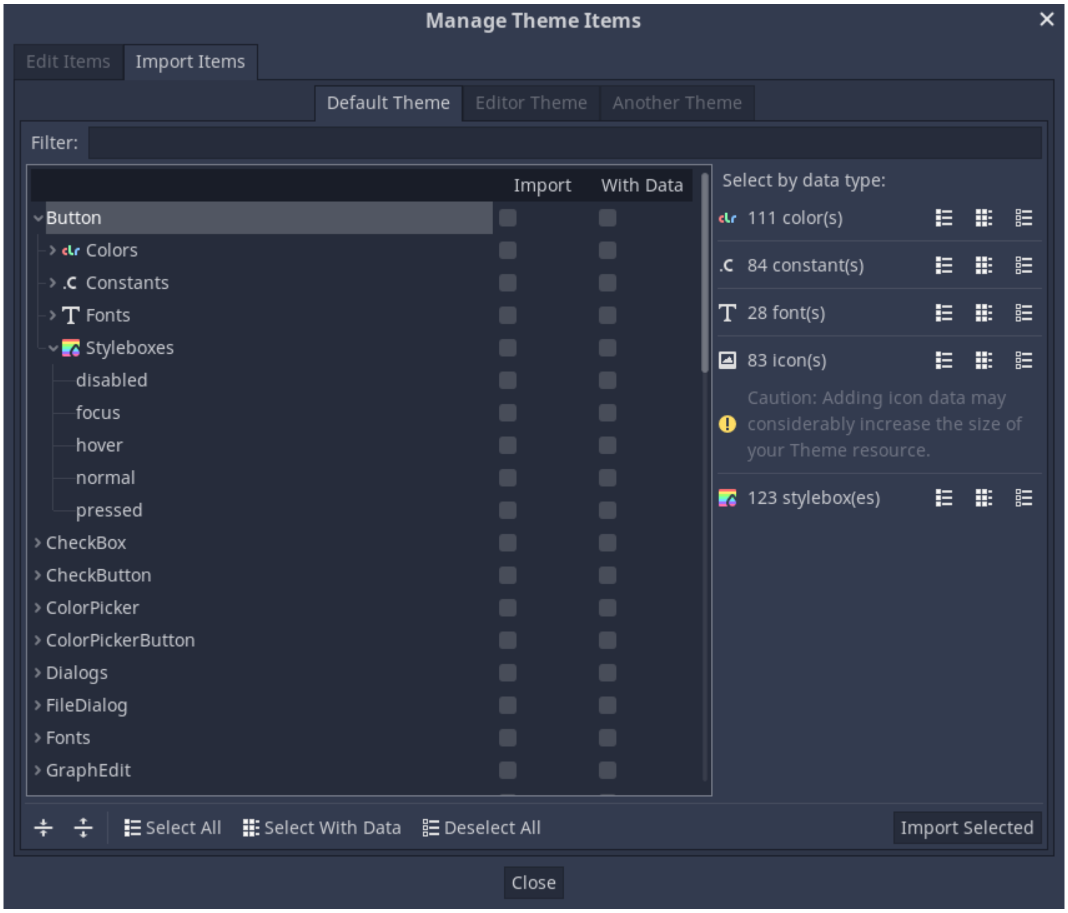
Manage and import items

Clicking the **Manage Items** button brings up the **Manage Theme Items** dialog. In the **Edit Items** tab you can view and add theme types, as well as view and edit the theme items of the selected type.


Manage Theme Items
Edit Items
Import Items
Types:
Button
Add Item:
Remove Items:
Styleboxes
normal
Add Type:
Add
Close

You can create, rename and remove individual theme items here by clicking the corresponding **Add X Item** and specifying their name. You can also mass delete theme items either by their data type (using the brush icon in the list) or by their quality. **Remove Class Items** will remove all built-in theme items you have customized for a control node type. **Remove Custom Items** will remove all the custom theme items for the selected type. Finally, **Remove All Items** will remove everything from the type.

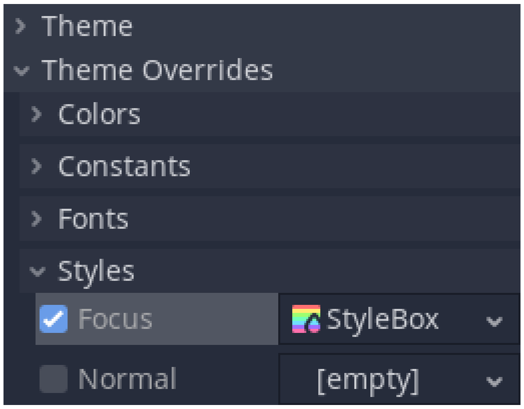
From the **Import Items** tab you can import theme items from other themes. You can import items from the default Godot theme, the Godot editor theme, or another custom theme. You can import individual or multiple items, and you can decide whether to copy or omit their data as well. There are several ways you can select and deselect the items, including by hand, by hierarchy, by data type, and everything. Opting to include the data will copy all theme items as they are to your theme. Omitting the data will create the items of the corresponding data type and name, but will leave them empty, creating a template of a theme in a way.


Manage Theme Items
Edit Items
Import Items
Default Theme
Editor Theme
Another Theme
Filter:
Import
With Data
Button
Colors
Constants
Fonts
Styleboxes
disabled
focus
hover
normal
pressed
CheckBox
CheckButton
ColorPicker
ColorPickerButton
Dialogs
FileDialog
Fonts
GraphEdit
Select by data type:
111 color(s)
84 constant(s)
28 font(s)
83 icon(s)
123 stylebox(es)
Select All
Select With Data
Deselect All
Import Selected
Close

Caution: Adding Icon data may considerably increase the size of your Theme resource.

Theme type variations

When designing a user interface there may be times when a Control node needs to have a different look than what is normally defined by a Theme. Every control node has theme property overrides, which allow you to redefine the styling for each individual UI element.



☑ Focus StyleBox

☐ Normal [empty]

This approach quickly becomes hard to manage, if you need to share the same custom look between several controls. Imagine that you use gray, blue, and red variants of Button throughout your project. Setting it up every time you add a new button element to your interface is a tedious task.

To help with the organization and to better utilize the power of themes you can use theme type variations. These work like normal theme types, but instead of being self-sufficient and standalone they extend another, base type.

Following the previous example, your theme can have some styles, colors, and fonts defined for the `Button` type, customizing the looks of every button element in your UI. To then have a gray, red, or blue button you would create a new type, e.g. `GrayButton`, and mark it as a variation of the base `Button` type.

Type variations can replace some aspects of the base type, but keep others. They can also define properties that the base style hasn't defined. For example, your `GrayButton` can override the `normal` style from the base `Button` and add `font_color` that `Button` has never defined. The control will use a combination of both types giving priority to the type variation.

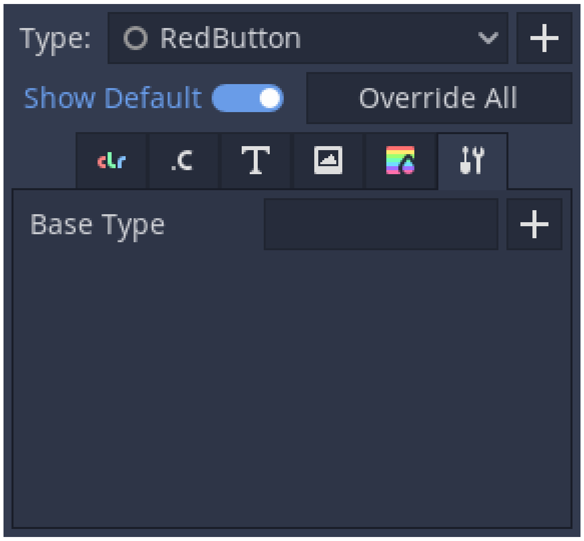
Note

The way controls resolve what theme items they use from each type and each theme is better described in the Customizing a project section of the "Introduction to GUI skinning" article.

Creating a type variation

To create a type variation open the theme editor, then click the plus icon next to the **Type** dropdown on the right side of the editor. Type in what you want to name your theme type variation in the text box, then click **Add Type**.

Below the **Type** dropdown are the property tabs. Switch to the tab with a wrench and screwdriver icon.



Click on the plus icon next to the **Base Type** field. You can select the base type there, which would typically be the name of a control node class (e.g., `Button`, `Label`, etc). Type variations can also chain and extend other type variations. This works in the same way control nodes inherit styling of their base class. For example, `CheckButton` inherits styles from `Button` because corresponding node types extend each other.

After you select the base type, you should now be able to see its properties on the other tabs in the theme editor. You can edit them as usual.

Using a type variation

Now that a type variation has been created you can apply it to your nodes. In the inspector dock, under the **Theme** property of a control node, you can find the **Theme Type Variation** property. It is empty by default, which means that only the base type has an effect on this node.

You can either select a type variation from a dropdown list, or input its name manually. Variations appear on the list only if the type variation belongs to the project-wide theme, which you can configure in the project settings. For any other case you have to input the name of the variation manually. Click on the pencil icon to the right. Then type in the name of the type variation and click the check mark icon or press enter. If a type variation with that name exists it will now be used by the node.

Using Fonts

Godot allows you to set specific fonts for different UI nodes.

There are three different places where you can setup font usage. The first is the theme editor. Choose the node you want to set the font for and select the font tab. The second is in the inspector for control nodes under **Theme Overrides > Fonts**. Lastly, in the inspector settings for themes under **Default Font**.

If no font override is specified anywhere, Open Sans [https://fonts.google.com/specimen/Open+Sans] SemiBold is used as the default project font.

Note

Since Godot 4.0, font sizes are no longer defined in the font itself but are instead defined in the node that uses the font. This is done in the **Theme Overrides > Font Sizes** section of the inspector.

This allows changing the font size without having to duplicate the font resource for every different font size.

There are 2 kinds of font files: *dynamic* (TTF/OTF/WOFF/WOFF2 formats) and *bitmap* (BMFont `.fnt` format or monospaced image). Dynamic fonts are the most commonly used option, as they can be resized and still look crisp at higher sizes. Thanks to their vector-based nature, they can also contain a lot more glyphs while keeping a reasonable file size compared to bitmap fonts. Dynamic fonts also support some advanced features that bitmap fonts cannot support, such as *ligatures* (several characters transforming into a single different design).

Tip

You can find freely licensed font files on websites such as Google Fonts [https://fonts.google.com/] and Font Library [https://fontlibrary.org/].

Fonts are covered by copyright. Double-check the license of a font before using it, as not all fonts allow commercial use without purchasing a license.

See also

You can see how fonts work in action using the BiDI and Font Features demo project [https://github.com/godotengine/godot-demo-projects/tree/master/gui/bidi_and_font_features].

Dynamic fonts

Godot supports the following dynamic font formats:

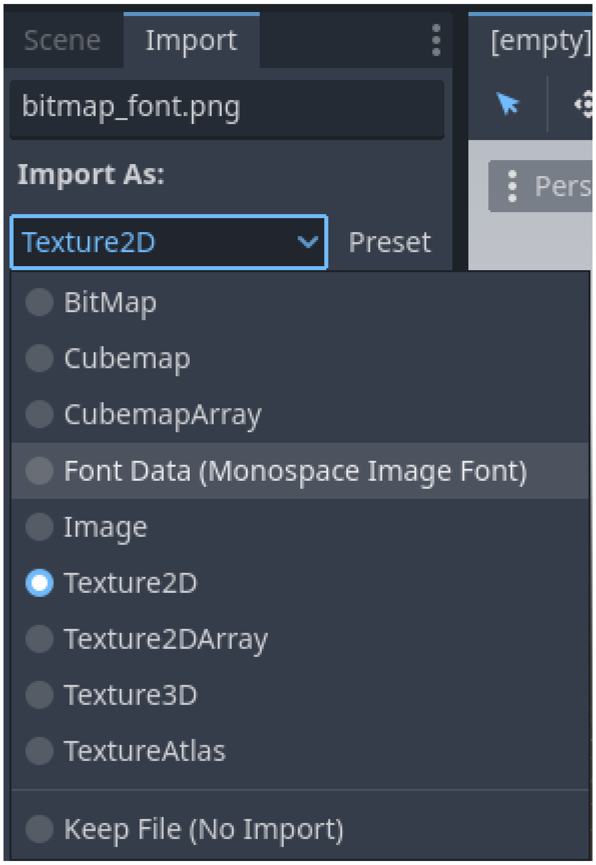
- TrueType Font or Collection (`.ttf`, `.ttc`)
- OpenType Font or Collection (`.otf`, `.otc`)
- Web Open Font Format 1 (`.woff`)
- Web Open Font Format 2 (`.woff2`, since Godot 3.5)

While `.woff` and especially `.woff2` tend to result in smaller file sizes, there is no universally "better" font format. In most situations, it's recommended to use the font format that was shipped on the font developer's website.

Bitmap fonts

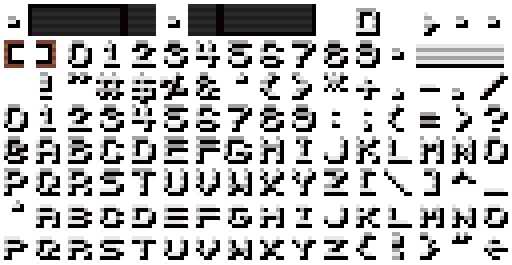
Godot supports the BMFont (`.fnt`) bitmap font format. This is a format created by the BMFont [https://www.angelcode.com/products/bmfont/] program. Many BMFont-compatible programs also exist, like BMGlyph [https://www.bmglyph.com/] or web-based fontcutter [https://github.com/fabienbk/fontcutter].

Alternatively, you can import any image to be used as a bitmap font. This is only supported for **monospaced** fonts (fonts where each character has the same width). To do so, select the image in the FileSystem dock, go to the Import dock, change its import type to **Font Data (Monospace Image Font)** then click **Reimport**:


Scene
Import
[empty]
bitmap_font.png
Import As:
Texture2D
Preset
Pers
BitMap
Cubemap
CubemapArray
Font Data (Monospace Image Font)
Image
Texture2D
Texture2DArray
Texture3D
TextureAtlas
Keep File (No Import)

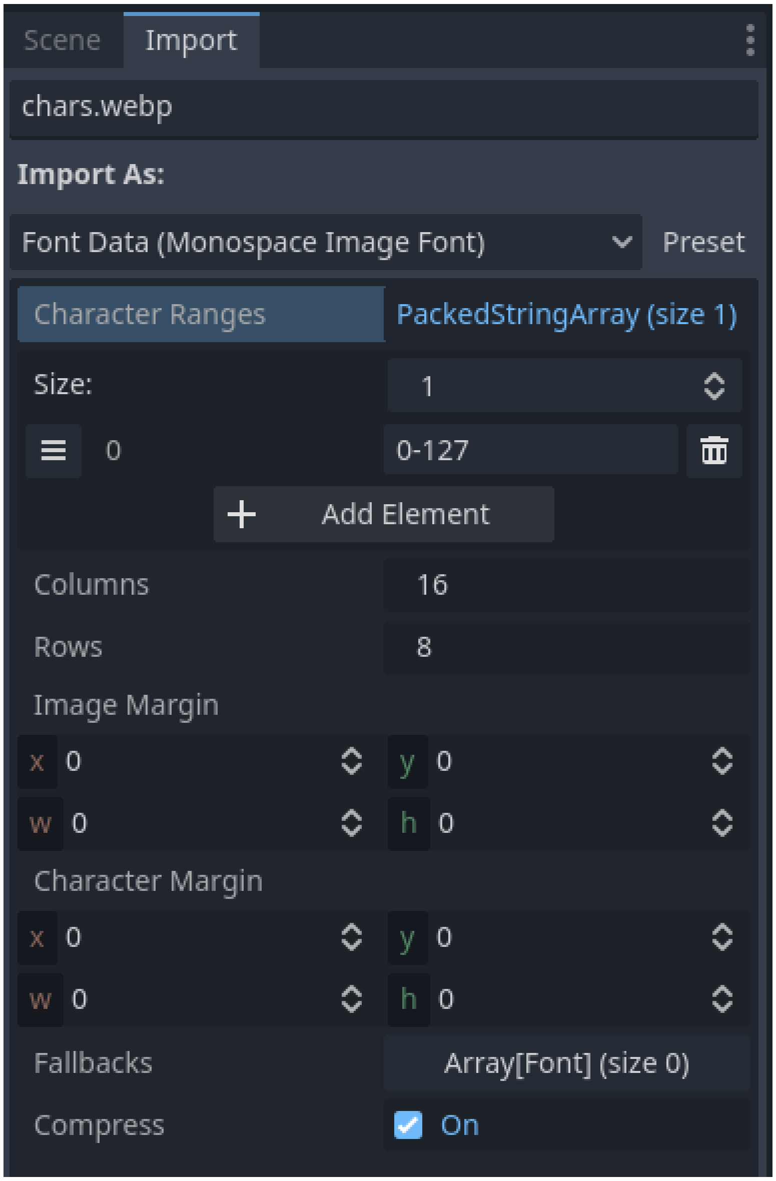
Changing import type to **Font Data (Monospace Image Font)**

The font's character set layout can be in any order, but orders that match standard Unicode are recommended as they'll require far less configuration to import. For example, the bitmap font below contains ASCII [https://en.wikipedia.org/wiki/ASCII] characters and follows standard ASCII ordering:



Credit: LibreQuake [https://github.com/MissLav/LibreQuake/blob/master/lq1/gfx-wad/CONCHARS.png] (scaled and cropped to exclude extended range)

The following import options can be used to import the above font image successfully:



chars.webp

Import As: Font Data (Monospace Image Font)

Character Ranges: PackedStringArray (size 1)

Size: 1

0: 0-127

Columns: 16

Rows: 8

Image Margin: x 0, y 0, w 0, h 0

Character Margin: x 0, y 0, w 0, h 0

Fallbacks: Array[Font] (size 0)

Compress: ☑ On

Import options to use for the above example font

The **Character Ranges** option is an array that maps each position on the image (in tile coordinates, not pixels). The font atlas is traversed from left to right and top to bottom. Characters can be specified with decimal numbers (`127`), hexadecimal numbers (`0x007f`) or between *single* quotes (`'~'`). Ranges can be specified with a hyphen between characters.

For instance, `0-127` (or `0x0000-0x007f`) denotes the full ASCII range. As another example, `' '-'~'` is equivalent to `32-127` and denotes the range of *printable* (visible) ASCII characters.

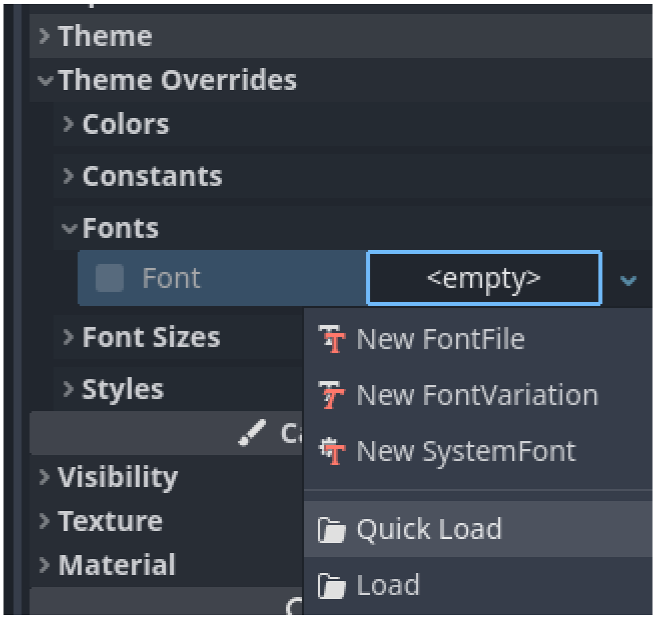
Make sure the **Character Ranges** option doesn't exceed the number of **Columns** × **Rows** defined. Otherwise, the font will fail to import.

If your font image contains margins not used for font glyphs (such as attribution information), try adjusting **Image Margin**. This is a margin applied only once around the whole image.

If your font image contains guides (in the form of lines between glyphs) or if spacing between characters appears incorrect, try adjusting **Character Margin**. This margin is applied for every imported glyph.

Loading a font file

To load a font file (dynamic or bitmap), use the resource dropdown's **Quick Load** or **Load** option next to a font property, then navigate to the font file in question:


Theme
Theme Overrides
Colors
Constants
Fonts
Font
<empty>
Font Sizes
Styles
Visibility
Texture
Material
New FontFile
New FontVariation
New SystemFont
Quick Load
Load

Loading a font file

You can also drag-and-drop a font file from the FileSystem dock to the inspector property that accepts a Font resource.

Warning

In Godot 4.0 and later, texture filter and repeat properties are defined in the location where the texture is used, rather than on the texture itself. This also applies to fonts (both dynamic fonts and bitmap fonts).

Fonts that have a pixel art appearance should have bilinear filtering disabled by changing the **Rendering > Textures > Canvas Textures > Default Texture Filter** project setting to **Nearest**.

The font size must also be an integer multiple of the design size (which varies on a per-font basis), and the Control node using the font must be scaled by an integer multiple as well. Otherwise, the font may look blurry. Font sizes in Godot are specified in pixels (px), not points (pt). Keep this in mind when comparing font sizes across different software.

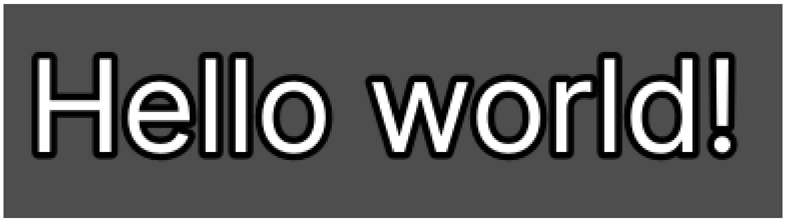
The texture filter mode can also be set on individual nodes that inherit from CanvasItem by setting CanvasItem.texture_filter.

Font outlines and shadows

Font outlines and shadows can be used to improve readability when the background color isn't known in advance. For instance, this is the case for HUD elements that are drawn over a 2D/3D scene.

Font outlines are available in most nodes that derive from Control, in addition to Label3D.

To enable outline for a font on a given node, configure the theme overrides **Font Outline Color** and **Outline Size** in the inspector. The result should look like this:

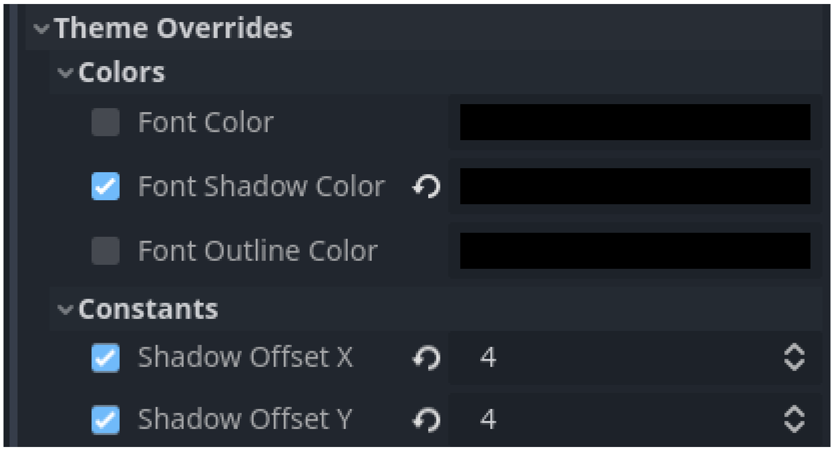


Font outline example

Note

If using a font with MSDF rendering, its **MSDF Pixel Range** import option be set to at least *twice* the value of the outline size for outline rendering to look correct. Otherwise, the outline may appear to be cut off earlier than intended.

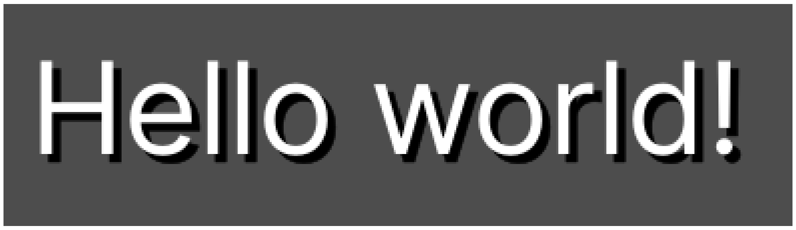
Support for font shadows is more limited: they are only available in Label and RichTextLabel. Additionally, font shadows always have a hard edge (but you can reduce their opacity to make them look more subtle). To enable font shadows on a given node, configure the **Font Shadow Color**, **Shadow Offset X**, and **Shadow Offset Y** theme overrides in a Label or RichTextLabel node accordingly:



Theme Overrides
Colors
☐ Font Color
☑ Font Shadow Color
☐ Font Outline Color
Constants
☑ Shadow Offset X 4
☑ Shadow Offset Y 4

Configuring font shadow in a Label node

The result should look like this:


Hello world!

Font shadow example

Tip

You can create local overrides to font display in Label nodes by creating a LabelSettings resource that you reuse across Label nodes. This resource takes priority over theme properties.

Advanced font features

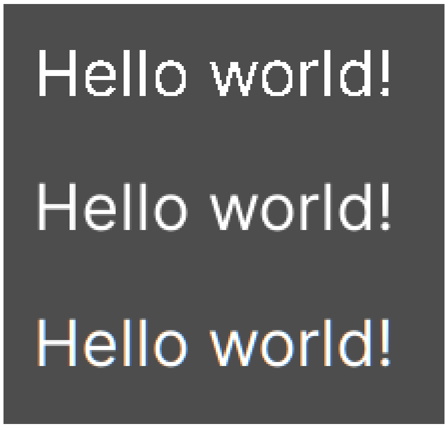
Antialiasing

You can adjust how the font should be smoothed out when rendering by adjusting *antialiasing* and *hinting*. These are different properties, with different use cases.

Antialiasing controls how glyph edges should be smoothed out when rasterizing the font. The default antialiasing method (**Grayscale**) works well on every display technology. However, at small sizes, grayscale antialiasing may result in fonts looking blurry.

The antialiasing sharpness can be improved by using LCD subpixel optimization, which exploits the subpixel patterns of most LCD displays by offsetting the font antialiasing on a per-channel basis (red/green/blue). The downside is that this can introduce "fringing" on edges, especially on display technologies that don't use standard RGB subpixels (such as OLED displays).

In most games, it's recommended to stick to the default **Grayscale** antialiasing. For non-game applications, LCD subpixel optimization is worth exploring.



Hello world!

Hello world!

Hello world!

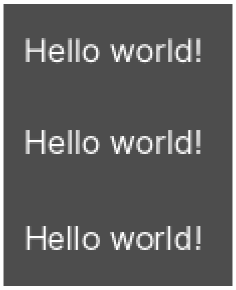
From top to bottom: Disabled, Grayscale, LCD Subpixel (RGB)

Note

Antialiasing cannot be changed on MSDF-rendered fonts – these are always rendered with grayscale antialiasing.

Hinting

Hinting controls how aggressively glyph edges should be snapped to pixels when rasterizing the font. **None** results in the smoothest appearance, which can make the font look blurry at small sizes. **Light** (default) is sharper by snapping glyph edges to pixels on the Y axis only, while **Full** is even sharper by snapping glyph edges to pixels on both X and Y axes. Depending on personal preference, you may prefer using one hinting mode over the other.



From top to bottom: None, Light, Full hinting

Note

If changing the hinting mode has no visible effect after clicking **Reimport**, it's usually because the font doesn't include hinting instructions. This can be resolved by looking for a version of the font file that includes hinting instructions, or enabling **Force Autohinter** in the Import dock. This will use FreeType [https://freetype.org/]'s autohinter to automatically add hinting instructions to the imported font.

Subpixel positioning

Subpixel positioning can be adjusted. This is a FreeType [https://freetype.org/] feature that allows glyphs to be rendered more closely to their intended form. The default setting of **Auto** automatically enables subpixel positioning at small sizes, but disables it at large font sizes to improve rasterization performance.

You can force the subpixel positioning mode to **Disabled**, **One half of a pixel** or **One quarter of a pixel**. **One quarter of a pixel** provides the best quality, at the cost of longer rasterization times.

Changing antialiasing, hinting and subpixel positioning has the most visible effect at smaller font sizes.

Warning

Fonts that have a pixel art appearance should have their subpixel positioning mode set to **Disabled**. Otherwise, the font may appear to have uneven pixel sizes.

This step is not required for bitmap fonts, as subpixel positioning is only relevant for dynamic fonts (which are usually made of vector elements).

Mipmaps

By default, fonts do not have mipmaps generated to reduce memory usage and speed up rasterization. However, this can cause downscaled fonts to become grainy. This can be especially noticeable with 3D text that doesn't have **Fixed Size** enabled. This can also occur when displaying text with a traditional rasterized (non-MSDF) font in a Control node that has its scale lower than `(1, 1)`.

After selecting a font in the FileSystem dock, you can enable the **Mipmaps** in the Import dock to improve downscaled font rendering appearance.

Mipmaps can be enabled on MSDF fonts as well. This can improve font rendering quality a little at smaller-than-default sizes, but MSDF fonts are already resistant to graininess out of the box.

MSDF font rendering

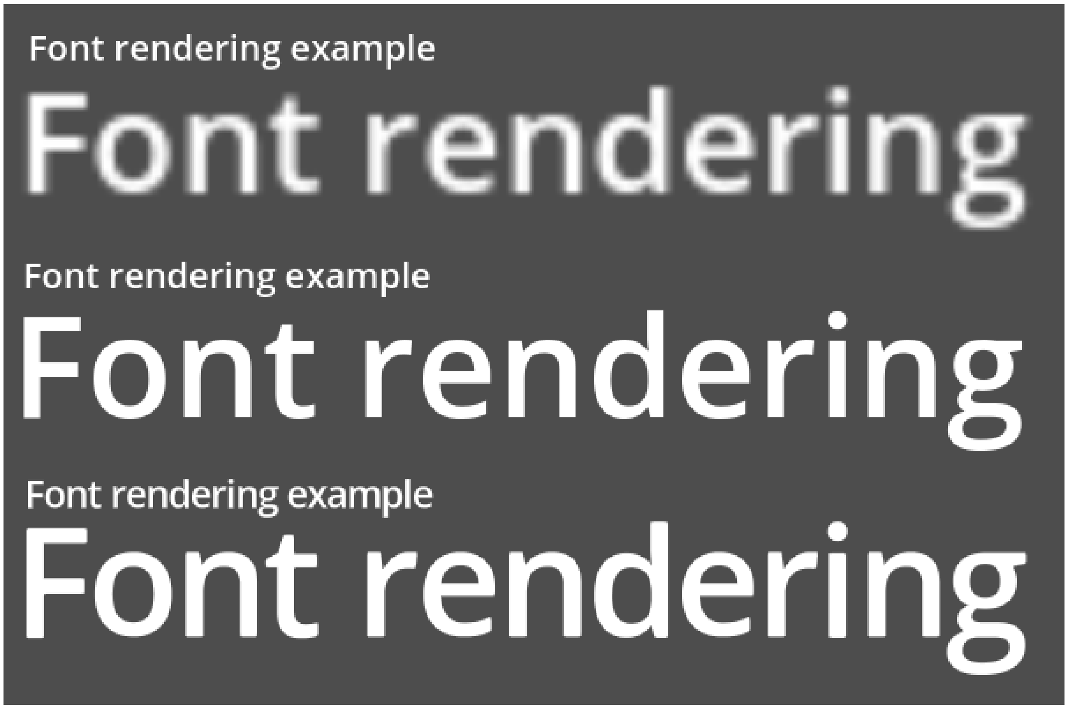
Multi-channel signed distance field (MSDF) font rendering allows rendering fonts at any size, without having to re-rasterize them when their size changes.

MSDF font rendering has 2 upsides over traditional font rasterization, which Godot uses by default:

- The font will always look crisp, even at huge sizes.
- There is less stuttering when rendering characters *at large font sizes* for the first time, as there is no rasterization performed.

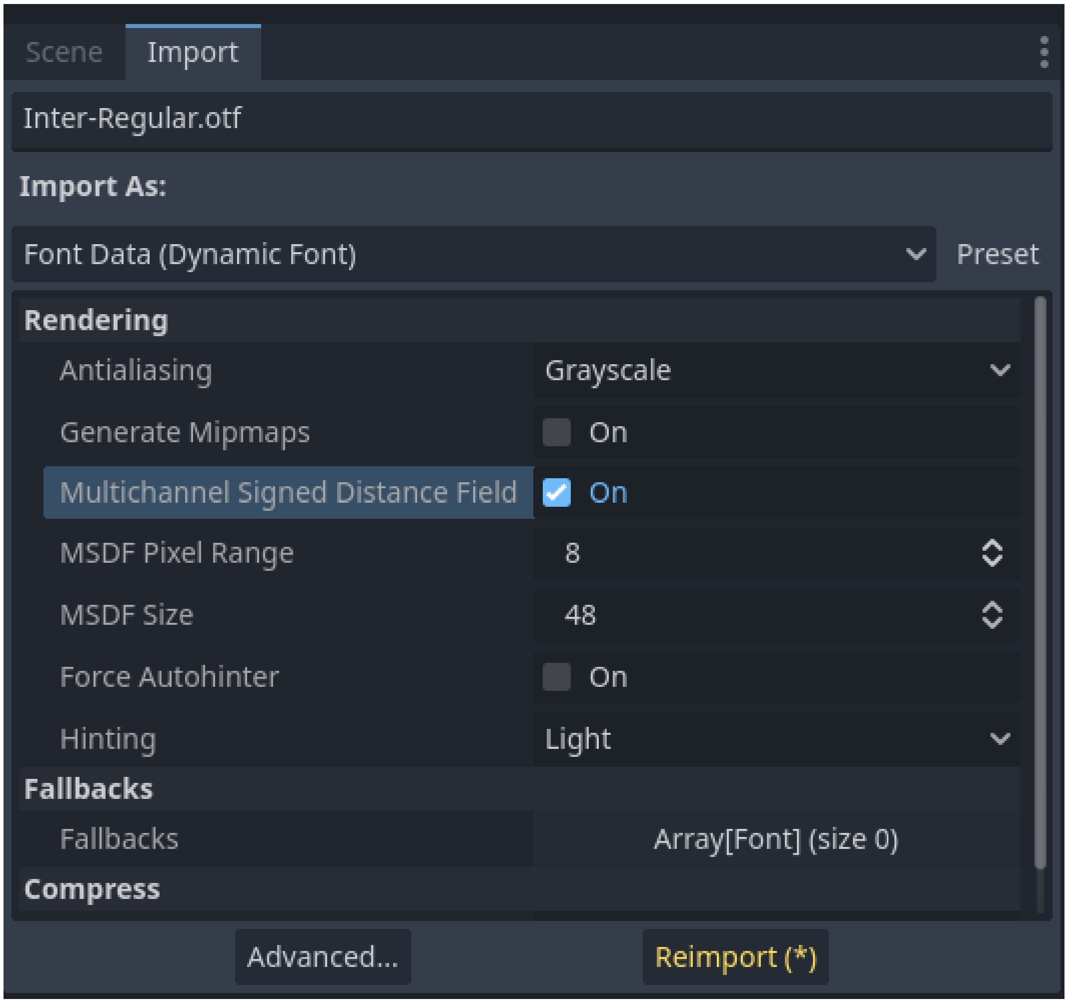
The downsides of MSDF font rendering are:

- Higher baseline cost for font rendering. This is usually not noticeable on desktop platforms, but it can have an impact on low-end mobile devices.
- Fonts at small sizes will not look as clear as rasterized fonts, due to the lack of hinting.
- Rendering new glyphs for the first time *at small font sizes* may be more expensive compared to traditional rasterized fonts. Font prerendering can be used to alleviate this.
- LCD subpixel optimization cannot be enabled for MSDF fonts.
- Fonts with self-intersecting outlines will not render correctly in MSDF mode. If you notice rendering issues on fonts downloaded from websites such as Google Fonts [https://fonts.google.com], try downloading the font from the font author's official website instead.


Font rendering example
Font rendering
Font rendering example
Font rendering
Font rendering example
Font rendering

Comparison of font rasterization methods. From top to bottom: rasterized without oversampling, rasterized with oversampling, MSDF

To enable MSDF rendering for a given font, select it in the FileSystem dock, go to the Import dock, enable **Multichannel Signed Distance Field**, then click **Reimport**:



Inter-Regular.otf

Import As: Font Data (Dynamic Font)

Rendering

Antialiasing	Grayscale
Generate Mipmaps	☐ On
Multichannel Signed Distance Field	☑ On
MSDF Pixel Range	8
MSDF Size	48
Force Autohinter	☐ On
Hinting	Light

Fallbacks

Fallbacks	Array[Font] (size 0)

Compress

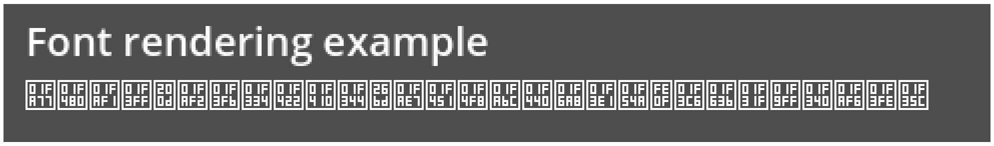
Enabling MSDF in the font's import options

Using emoji

Godot has limited support for emoji fonts:

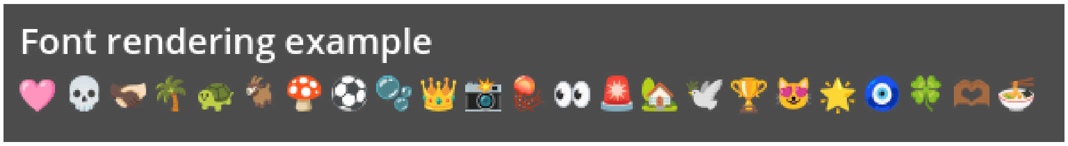
- CBDT/CBLC (embedded PNGs) and SVG emoji fonts are supported.
- COLR/CPAL emoji fonts (custom vector format) are **not** supported.
- EMJC bitmap image compression (used by iOS' system emoji font) is **not** supported. This means that to support emoji on iOS, you must use a custom font that uses SVG or PNG bitmap compression instead.

For Godot to be able to display emoji, the font used (or one of its fallbacks) needs to include them. Otherwise, emoji won't be displayed and placeholder "tofu" characters will appear instead:


Font rendering example

Default appearance when trying to use emoji in a label

After adding a font to display emoji such as Noto Color Emoji [https://fonts.google.com/noto/specimen/Noto+Color+Emoji], you get the expected result:


Font rendering example

Correct appearance after adding an emoji font to the label

To use a regular font alongside emoji, it's recommended to specify a fallback font that points to the emoji font in the regular font's advanced import options. If you wish to use the default project font while displaying emoji, leave the **Base Font** property in FontVariation empty while adding a font fallback pointing to the emoji font:

Tip

Emoji fonts are quite large in size, so you may want to load a system font to provide emoji glyphs rather than bundling it with your project. This allows providing full emoji support in your project without increasing the size of its exported PCK. The downside is that emoji will look different depending on the platform, and loading system fonts is not supported on all platforms.

It's possible to use a system font as a fallback font too.

Using icon fonts

Tools like Fontello [https://fontello.com/] can be used to generate font files containing vectors imported from SVG files. This can be used to render custom vector elements as part of your text, or to create extruded 3D icons with 3D text and TextMesh.

Note

Fontello currently does not support creating multicolored fonts (which Godot can render). As of November 2022, support for multicolored fonts in icon font generation tools remains scarce.

Depending on your use cases, this may lead to better results compared to using the `img` tag in RichTextLabel. Unlike bitmap images (including SVGs which are rasterized on import by Godot), true vector data can be resized to any size without losing quality.

After downloading the generated font file, load it in your Godot project then specify it as a custom font for a Label, RichTextLabel or Label3D node. Switch over to the Fontello web interface, then copy the character by selecting it then pressing `Ctrl + C` (`Cmd + C` on macOS). Paste the character in the **Text** property of your Label node. The character will appear as a placeholder glyph in the inspector, but it should appear correctly in the 2D/3D viewport.

To use an icon font alongside a traditional font in the same Control, you can specify the icon font as a fallback. This works because icon fonts use the Unicode *private use area*, which is reserved for use by custom fonts and doesn't contain standard glyphs by design.

Note

Several modern icon fonts such as Font Awesome 6 [https://fontawesome.com/download] have a desktop variant that uses *ligatures* to specify icons. This allows you to specify icons by entering their name directly in the **Text** property of any node that can display fonts. Once the icon's name is fully entered as text (such as `house`), it will be replaced by the icon.

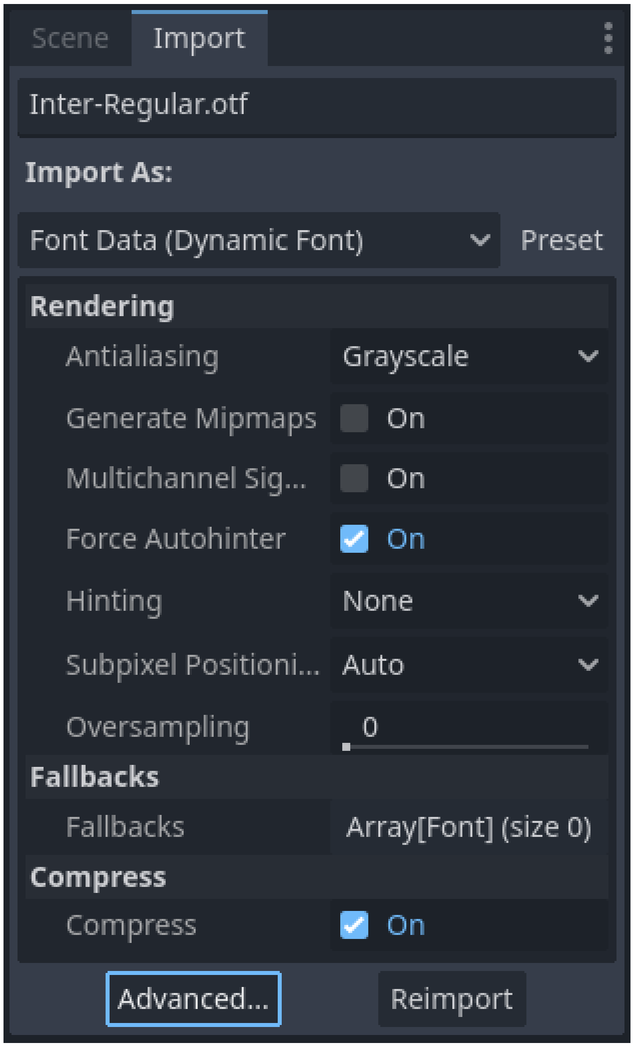
While easier to use, this approach cannot be used with font fallbacks as the main font's characters will take priority over the fallback font's ligatures.

Font fallbacks

Godot supports defining one or more fallbacks when the main font lacks a glyph to be displayed. There are 2 main use cases for defining font fallbacks:

- Use a font that only supports Latin character sets, but use another font to be able to display text another character set such as Cyrillic.
- Use a font to render text, and another font to render emoji or icons.

Open the Advanced Import Settings dialog by double-clicking the font file in the FileSystem dock. You can also select the font in the FileSystem dock, go to the Import dock then choose **Advanced…** at the bottom:



Inter-Regular.otf

Import As:

Font Data (Dynamic Font)

Rendering

Antialiasing: Grayscale

Generate Mipmaps: ☐ On

Multichannel Sig...: ☐ On

Force Autohinter: ☑ On

Hinting: None

Subpixel Positioni...: Auto

Oversampling: 0

Fallbacks

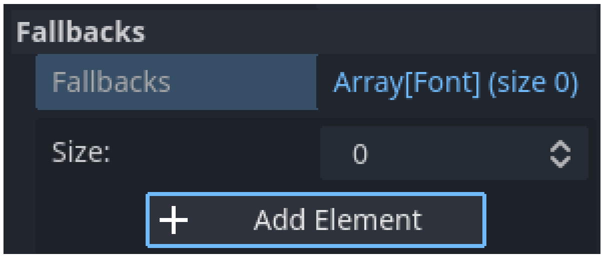
Fallbacks: Array[Font] (size 0)

Compress

Compress: ☑ On

Import dock

In the dialog that appears, look for **Fallbacks** section on the sidebar on the right, click the **Array[Font] (size 0)** text to expand the property, then click **Add Element**:



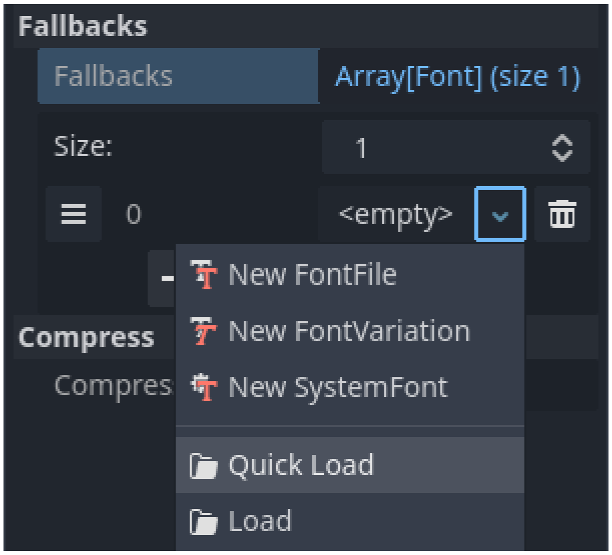
Fallbacks

Fallbacks: Array[Font] (size 0)

Size: 0

Adding font fallback

Click the dropdown arrow on the new element, then choose a font file using the **Quick Load** or **Load** options:


Fallbacks
Fallbacks
Array[Font] (size 1)
Size:
1
0
<empty>
New FontFile
New FontVariation
New SystemFont
Quick Load
Load
Compress
Compres

Loading font fallback

It is possible to add fallback fonts while using the default project font. To do so, leave the **Base Font** property empty while adding one or more font fallbacks.

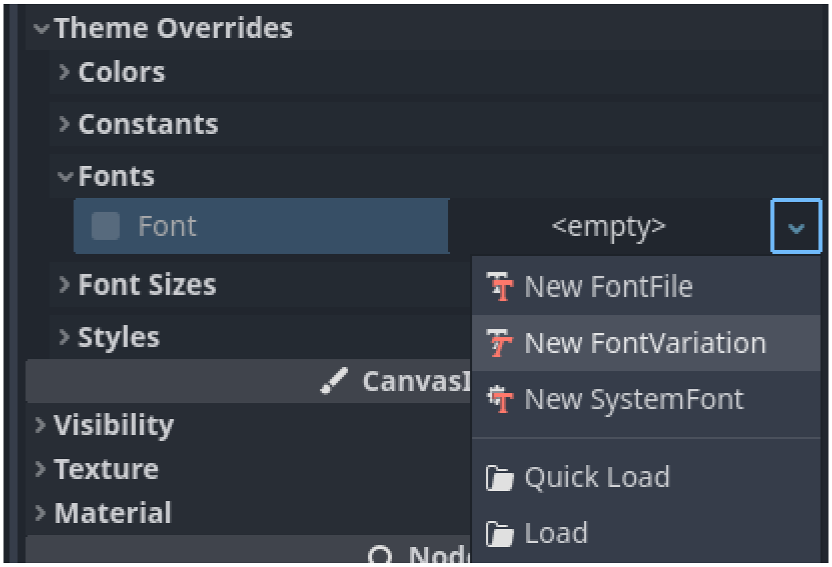
Note

Font fallbacks can also be defined on a local basis similar to OpenType font features, but this is not covered here for brevity reasons.

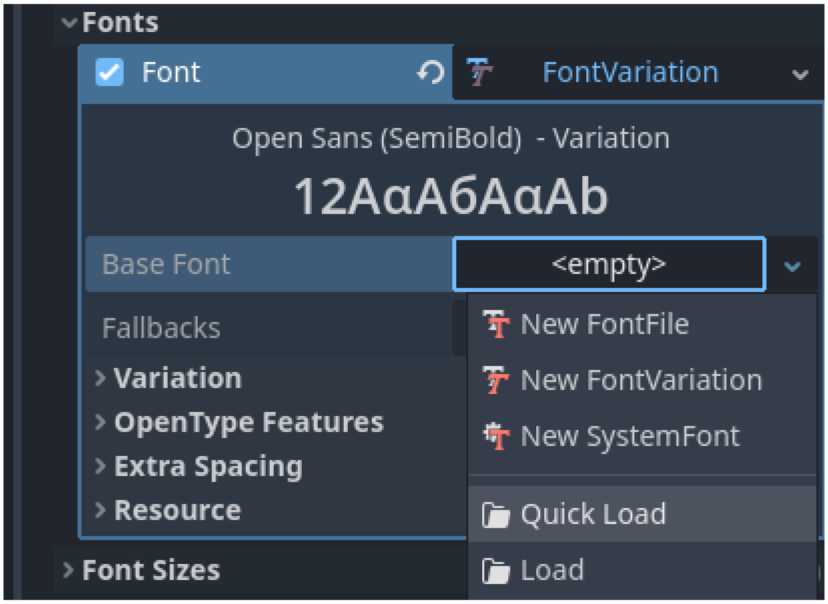
Variable fonts

Godot has full support for variable fonts [https://variablefonts.io/], which allow you to use a single font file to represent various font weights and styles (regular, bold, italic, …). This must be supported by the font file you're using.

To use a variable font, create a [FontVariation]() resource in the location where you intend to use the font, then load a font file within the FontVariation resource:

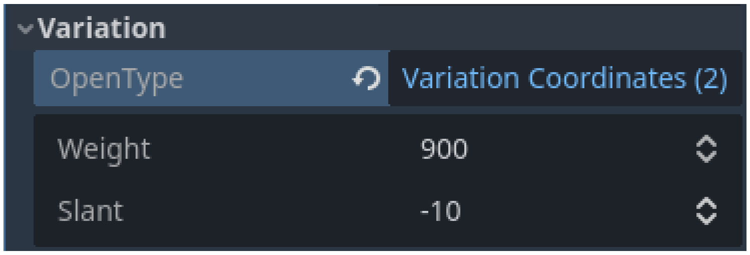

Theme Overrides
Colors
Constants
Fonts
Font
<empty>
New FontFile
New FontVariation
New SystemFont
Quick Load
Load
Font Sizes
Styles
CanvasI
Visibility
Texture
Material
Nod

Creating a FontVariation resource


Fonts
Font
FontVariation
Open Sans (SemiBold) - Variation
12AaA6AaAb
Base Font
<empty>
Fallbacks
Variation
OpenType Features
Extra Spacing
Resource
New FontFile
New FontVariation
New SystemFont
Quick Load
Load
Font Sizes

Loading a font file into the FontVariation resource

Scroll down to the FontVariation's **Variation** section, then click the **Variation Coordinates** text to expand the list of axes that can be adjusted:


Variation

OpenType: Variation Coordinates (2)

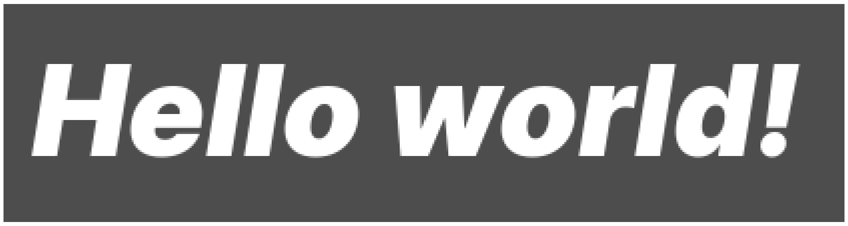
Weight: 900

Slant: -10

List of variation axes

The set of axes you can adjust depends on the font loaded. Some variable fonts only support one axis of adjustment (typically *weight* or *slant*), while others may support multiple axes of adjustment.

For example, here's the [Inter V](https://rsms.me/inter/) [https://rsms.me/inter/] font with a *weight* of `900` and a *slant* of `-10`:


Hello world!

Variable font example (Inter V)

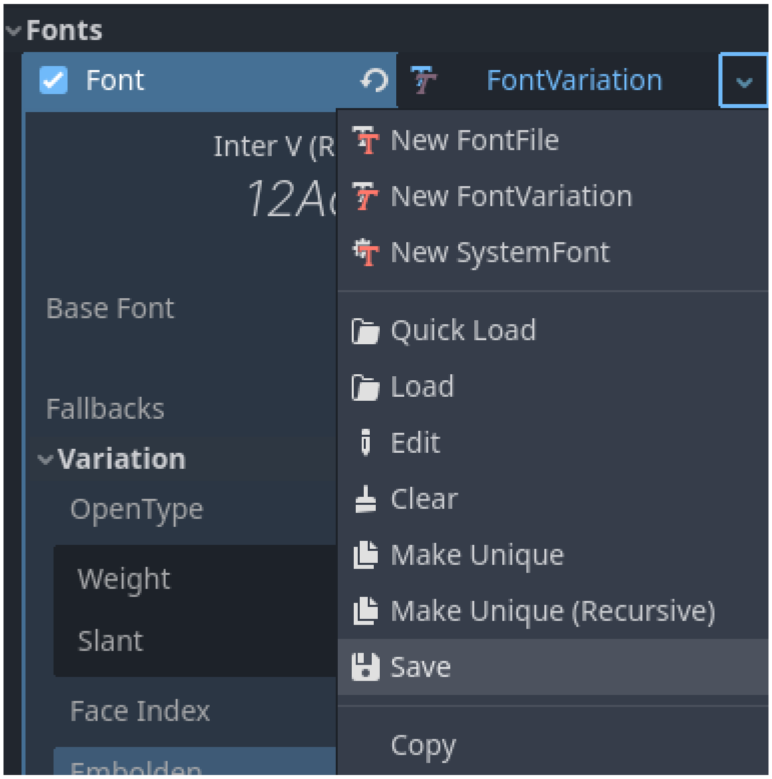
Tip

While variable font axis names and scales aren't standardized, some common conventions are usually followed by font designers. The *weight* axis is standardized in OpenType to work as follows:

Axis value	Effective font weight
`100`	Thin (Hairline)

Axis value	Effective font weight
`200`	Extra Light (Ultra Light)
`300`	Light
`400`	**Regular (Normal)**
`500`	Medium
`600`	Semi-Bold (Demi-Bold)
`700`	**Bold**
`800`	Extra Bold (Ultra Bold)
`900`	Black (Heavy)
`950`	Extra Black (Ultra Black)

You can save the FontVariation to a `.tres` resource file to reuse it in other places:


Fonts
Font
FontVariation
Inter V (R
12A
Base Font
Fallbacks
Variation
OpenType
Weight
Slant
Face Index
Embolden
New FontFile
New FontVariation
New SystemFont
Quick Load
Load
Edit
Clear
Make Unique
Make Unique (Recursive)
Save
Copy

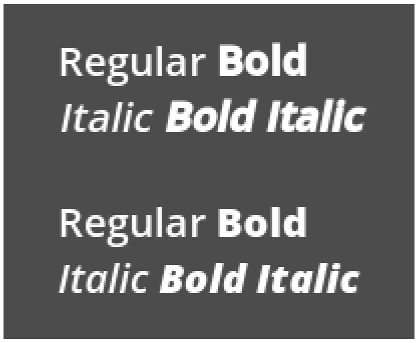
Saving FontVariation to an external resource file

Faux bold and italic

When writing text in bold or italic, using font variants specifically designed for this looks better. Spacing between glyphs will be more consistent when using a bold font, and certain glyphs' shapes may change entirely in italic variants (compare "a" and *"a"*).

However, real bold and italic fonts require shipping more font files, which increases distribution size. A single variable font file can also be used, but this file will be larger than a single non-variable font. While file size is usually not an issue for desktop projects, it can be a concern for mobile/web projects that strive to keep distribution size as low as possible.

To allow bold and italic fonts to be displayed without having to ship additional fonts (or use a variable font that is larger in size), Godot supports *faux* bold and italic.



Faux bold/italic (top), real bold/italic (bottom). Normal font used: Open Sans SemiBold

Faux bold and italic is automatically used in RichTextLabel's bold and italic tags if no custom fonts are provided for bold and/or italic.

To use faux bold, create a FontVariation resource in a property where a Font resource is expected. Set **Variation > Embolden** to a positive value to make a font bolder, or to a negative value to make it less bold. Recommended values are between `0.5` and `1.2` depending on the font.

Faux italic is created by skewing the text, which is done by modifying the per-character transform. This is also provided in FontVariation using the **Variation > Transform** property. Setting the `yx` component of the character transform to a positive value will italicize the text. Recommended values are between `0.2` and `0.4` depending on the font.

Adjusting font spacing

For stylistic purposes or for better readability, you may want to adjust how a font is presented in Godot.

Create a FontVariation resource in a property where a Font resource is expected. There are 4 properties available in the **Variation >**

Extra Spacing section, which accept positive and negative values:

- **Glyph:** Additional spacing between every glyph.
- **Space:** Additional spacing between words.
- **Top:** Additional spacing above glyphs. This is used for multiline text, but also to calculate the minimum size of controls such as Label and Button.
- **Top:** Additional spacing below glyphs. This is used for multiline text, but also to calculate the minimum size of controls such as Label and Button.

The **Variation > Transform** property can also be adjusted to stretch characters horizontally or vertically. This is specifically done by adjusting the `xx` (horizontal scale) and `yy` (vertical scale) components. Remember to adjust glyph spacing to account for any changes, as glyph transform doesn't affect how much space each glyph takes in the text. Non-uniform scaling of this kind should be used sparingly, as fonts are generally not designed to be displayed with stretching.

OpenType font features

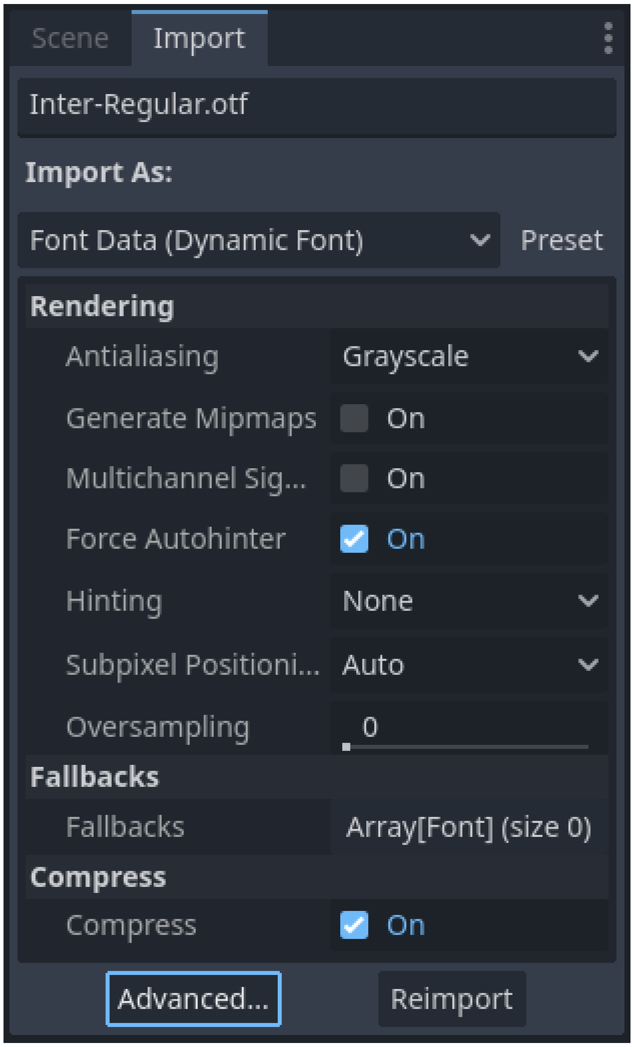
Godot supports enabling OpenType font features, which are a standardized way to define alternate characters that can be toggled without having to swap font files entirely. Despite being named OpenType font features, these are also supported in TrueType (`.ttf`) and WOFF/WOFF2 font files.

Support for OpenType features highly depends on the font used. Some fonts don't support any OpenType features, while other fonts can support dozens of toggleable features.

There are 2 ways to use OpenType font features:

Globally on a font file

Open the Advanced Import Settings dialog by double-clicking the font file in the FileSystem dock. You can also select the font in the FileSystem dock, go to the Import dock then choose **Advanced…** at the bottom:



Inter-Regular.otf

Import As:

Font Data (Dynamic Font)

Rendering

Antialiasing	Grayscale
Generate Mipmaps	☐ On
Multichannel Sig...	☐ On
Force Autohinter	☑ On
Hinting	None
Subpixel Positioni...	Auto
Oversampling	0

Fallbacks

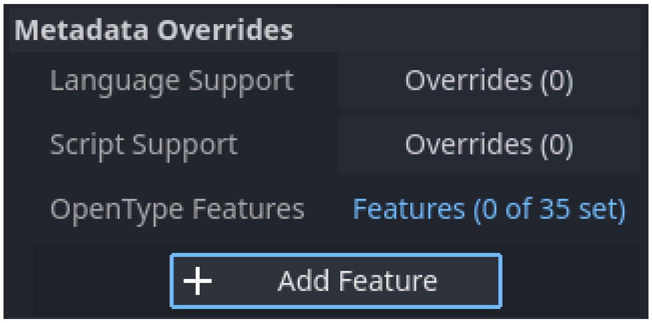
Fallbacks	Array[Font] (size 0)

Compress

Compress	☑ On

Import dock

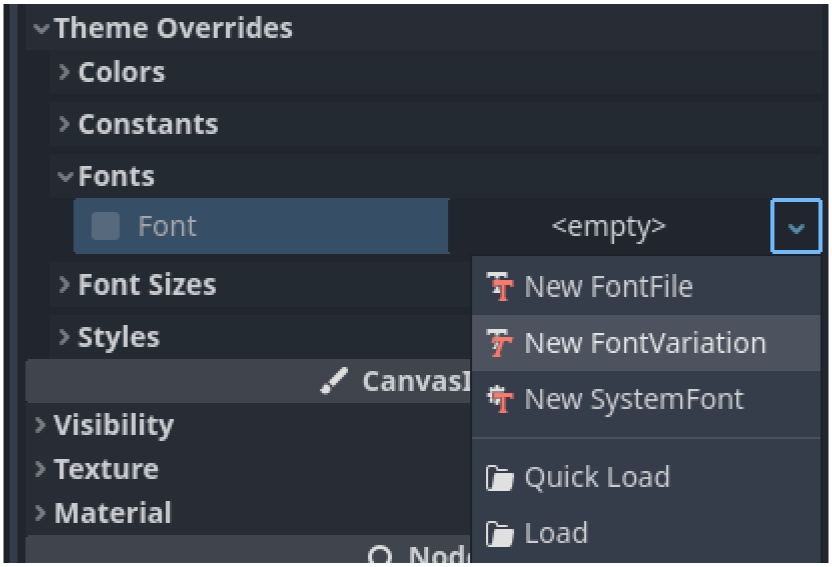
In the dialog that appears, look for the **Metadata Overrides > OpenType Features** section on the sidebar on the right, click the **Features (0 of N set)** text to expand the property, then click **Add Feature**:


Metadata Overrides
Language Support
Overrides (0)
Script Support
Overrides (0)
OpenType Features
Features (0 of 35 set)
Add Feature

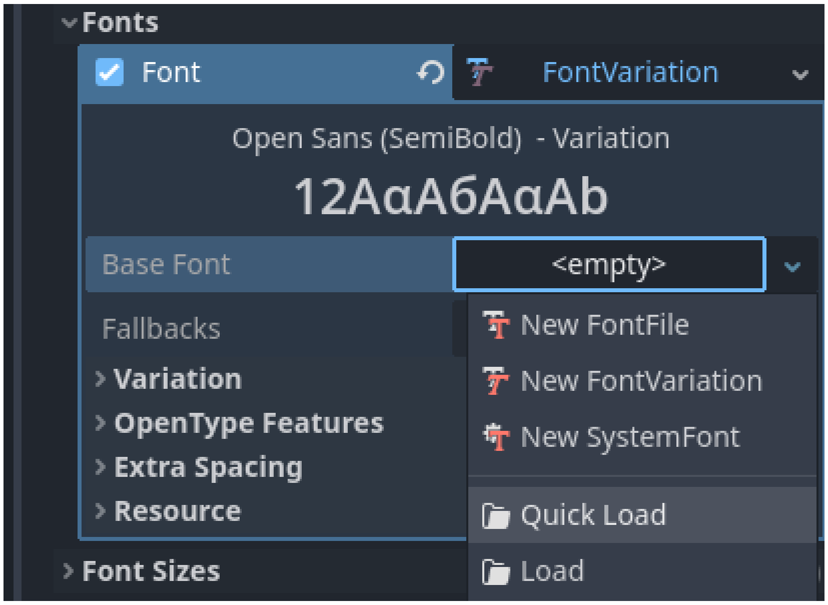
OpenType feature overrides in Advanced Import Settings

In a specific font usage (FontVariation)

To use a font feature, create a FontVariation resource like you would do for a variable font, then load a font file within the FontVariation resource:

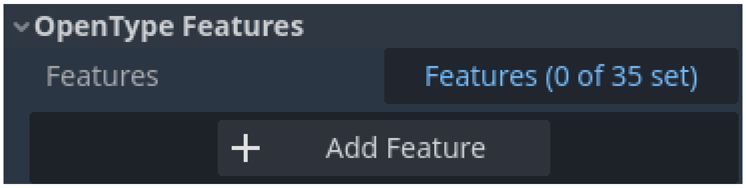

Theme Overrides
Colors
Constants
Fonts
Font
<empty>
Font Sizes
Styles
CanvasI
Visibility
Texture
Material
Nod
New FontFile
New FontVariation
New SystemFont
Quick Load
Load

Creating a FontVariation resource


Fonts
Font
FontVariation
Open Sans (SemiBold) - Variation
12AɑA6AɑAb
Base Font
<empty>
Fallbacks
Variation
OpenType Features
Extra Spacing
Resource
Font Sizes
New FontFile
New FontVariation
New SystemFont
Quick Load
Load

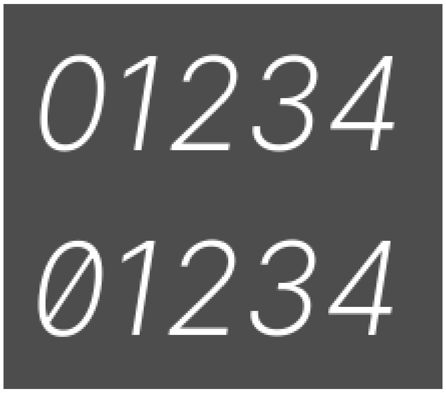
Loading a font file into a FontVariation resource

Scroll down to the FontVariation's **OpenType Features** section, click the **Features (0 of N set)** text to expand the property, then click **Add Feature** and select the desired feature in the dropdown:


OpenType Features
Features
Features (0 of 35 set)
Add Feature

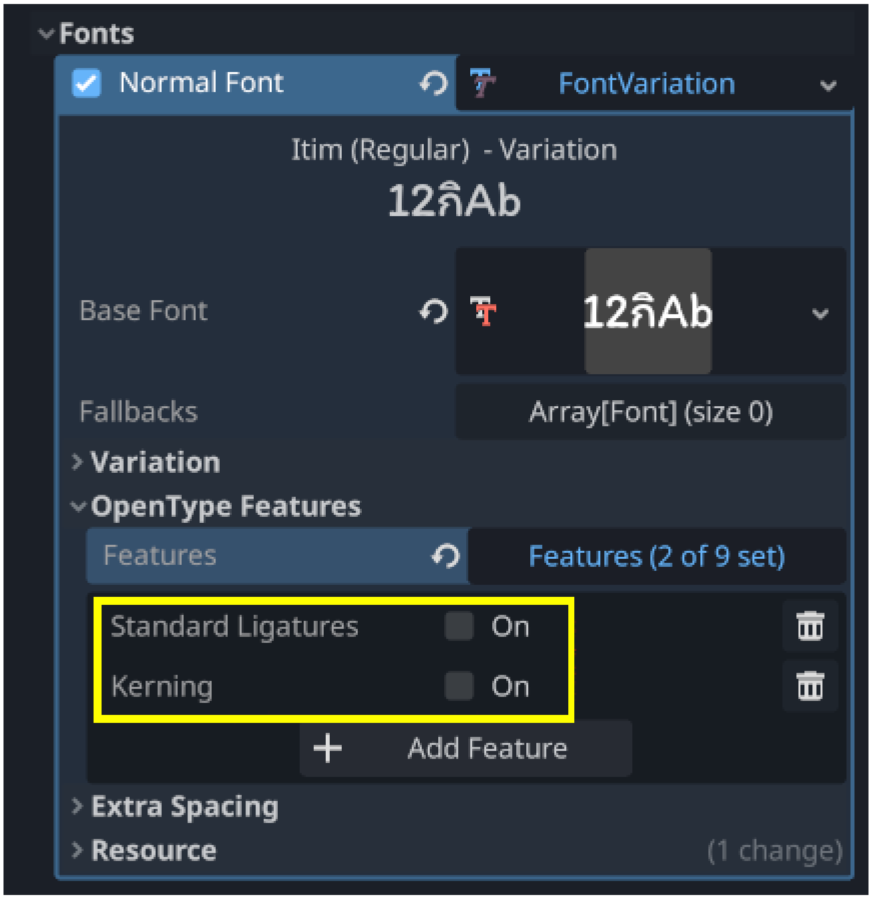
Specifying OpenType features in a FontVariation resource

For example, here's the Inter [https://rsms.me/inter/] font without the *Slashed Zero* feature (top), then with the *Slashed Zero* OpenType feature enabled (bottom):



OpenType feature comparison (Inter)

You can disable ligatures and/or kerning for a specific font by adding OpenType features, then unchecking them in the inspector:



Fonts

☑ Normal Font FontVariation

Itim (Regular) - Variation

12กิAb

Base Font 12กิAb

Fallbacks Array[Font] (size 0)

Variation

OpenType Features

Features Features (2 of 9 set)

Standard Ligatures ☐ On

Kerning ☐ On

Extra Spacing

Resource (1 change)

Disabling ligatures and kerning for a font

System fonts

Warning

Loading system fonts is only supported on Windows, macOS, Linux, Android and iOS.

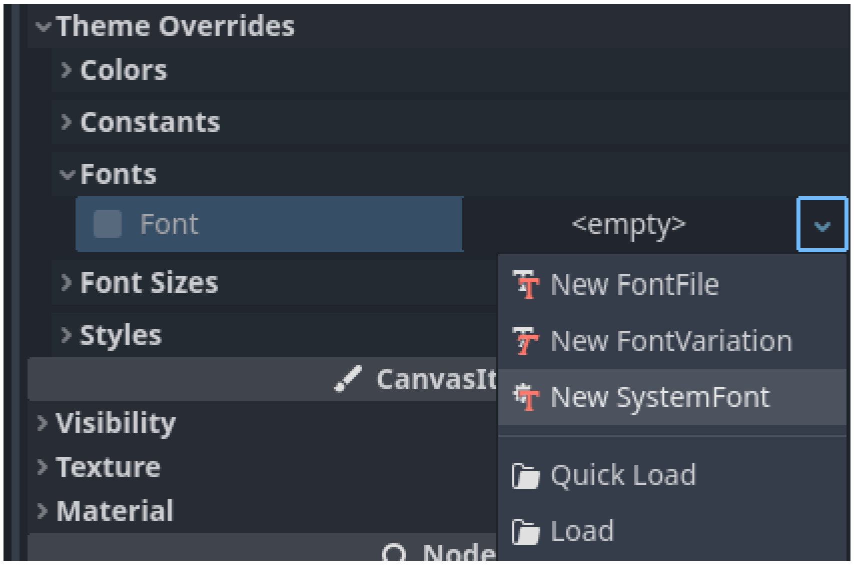
However, loading system fonts on Android is unreliable as there is no official API for doing so. Godot has to rely on parsing system configuration files, which can be modified by third-party Android vendors. This may result in non-functional system font loading.

System fonts are a different type of resource compared to imported fonts. They are never actually imported into the project, but are loaded at run-time. This has 2 benefits:

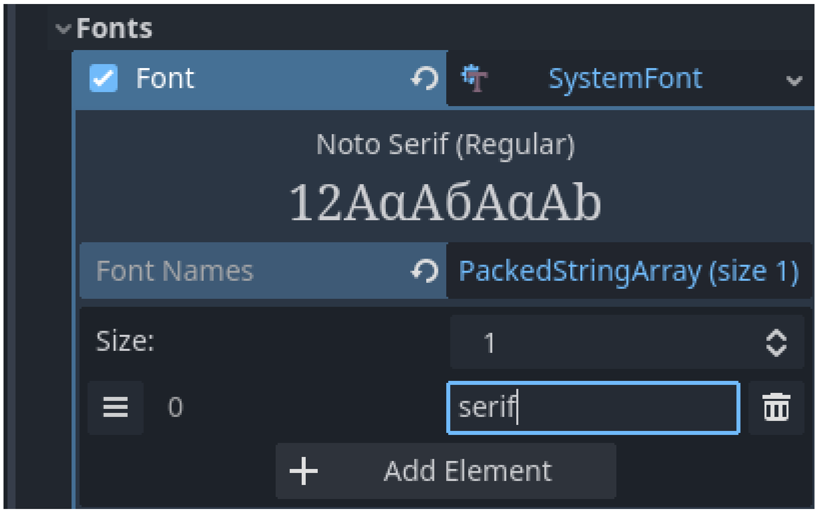
- The fonts are not included within the exported PCK file, leading to a smaller file size for the exported project.
- Since fonts are not included with the exported project, this avoids licensing issues that would occur if proprietary system fonts were distributed alongside the project.

The engine automatically uses system fonts as fallback fonts, which makes it possible to display CJK characters and emoji without having to load a custom font. There are some restrictions that apply though, as mentioned in the Using emoji section.

Create a SystemFont resource in the location where you desire to use the system font:


Theme Overrides
Colors
Constants
Fonts
Font
<empty>
Font Sizes
Styles
CanvasIt
Visibility
Texture
Material
Node
New FontFile
New FontVariation
New SystemFont
Quick Load
Load

Creating a SystemFont resource


Fonts
Font
SystemFont
Noto Serif (Regular)
12AαAбAαAb
Add Element

Font Names: PackedStringArray (size 1)

Size: 1

0: serif

Specifying a font name to use in a SystemFont resource

You can either specify one or more font names explicitly (such as `Arial`), or specify the name of a font *alias* that maps to a "standard" default font for the system:

Font alias	**Windows**	**macOS/iOS**	**Linux**	**Android**
`sans-serif`	Arial	Helvetica	*Handled by fontconfig*	Roboto / Noto Sans
`serif`	Times New Roman	Times	*Handled by fontconfig*	Noto Serif
`monospace`	Courier New	Courier	*Handled by fontconfig*	Droid Sans Mono
`cursive`	Comic Sans MS	Apple Chancery	*Handled by fontconfig*	Dancing Script
`fantasy`	Gabriola	Papyrus	*Handled by fontconfig*	Droid Sans Mono

On Android, Roboto is used for Latin/Cyrillic text and Noto Sans is used for other languages' glyphs such as CJK. On third-party Android distributions, the exact font selection may differ.

If specifying more than one font, the first font that is found on the system will be used (from top to bottom). Font names and aliases are case-insensitive on all platforms.

Like for font variations, you can save the SystemFont arrangement to a resource file to reuse it in other places.

Remember that different system fonts have different metrics, which means that text that can fit within a rectangle on one platform may not be doing so on another platform. Always reserve some additional space during development so that labels can extend further if needed.

Note

Unlike Windows and macOS/iOS, the set of default fonts shipped on Linux depends on the distribution. This means that on different Linux distributions, different fonts may be displayed for a given system font name or alias.

It is also possible to load fonts at runtime even if they aren't installed on the system. See Runtime loading and saving for details.

Font prerendering

When using traditional rasterized fonts, Godot caches glyphs on a per-font and per-size basis. This reduces stuttering, but it can still occur the first time a glyph is displayed when running the project. This can be especially noticeable at higher font sizes or on mobile devices.

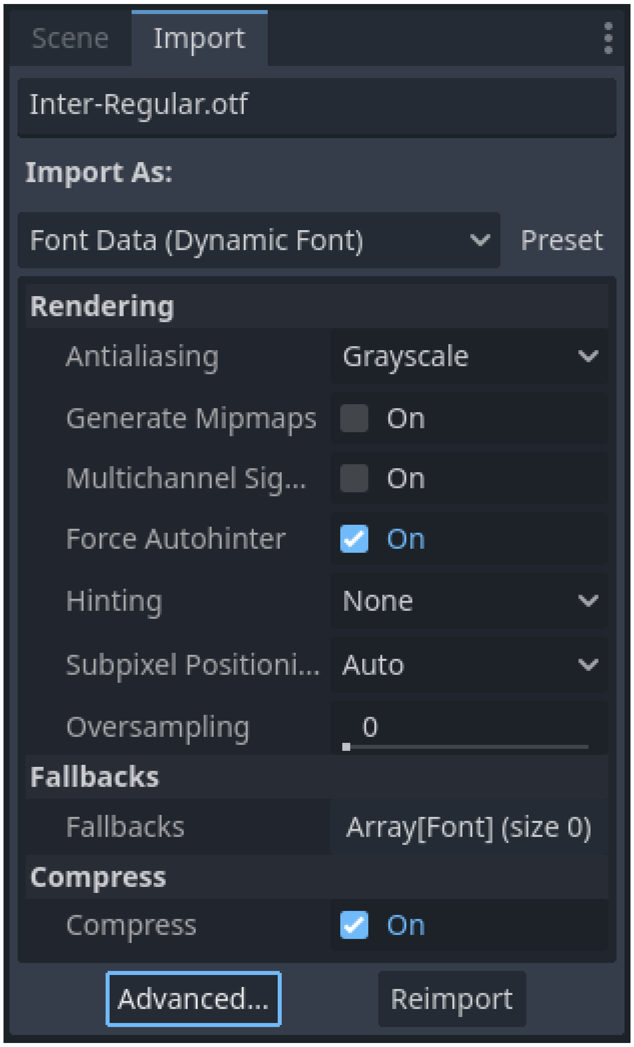
When using MSDF fonts, they only need to be rasterized once to a special signed distance field texture. This means caching can be done purely on a per-font basis, without taking the font size into consideration. However, the initial rendering of MSDF fonts is slower compared to a traditional rasterized font at a medium size.

To avoid stuttering issues related to font rendering, it is possible to *prerender* certain glyphs. This can be done for all glyphs you intend to use (for optimal results), or only for common glyphs that are most likely to appear during gameplay (to reduce file size). Glyphs that aren't pre-rendered will be rasterized on-the-fly as usual.

Note

In both cases (traditional and MSDF), font rasterization is done on the CPU. This means that the GPU performance doesn't affect how long it takes for fonts to be rasterized.

Open the Advanced Import Settings dialog by double-clicking the font file in the FileSystem dock. You can also select the font in the FileSystem dock, go to the Import dock then choose **Advanced…** at the bottom:


Scene
Import
Advanced...
Reimport

Inter-Regular.otf

Import As: Font Data (Dynamic Font) Preset

Rendering

Antialiasing: Grayscale

Generate Mipmaps: ☐ On

Multichannel Sig...: ☐ On

Force Autohinter: ☑ On

Hinting: None

Subpixel Positioni...: Auto

Oversampling: 0

Fallbacks

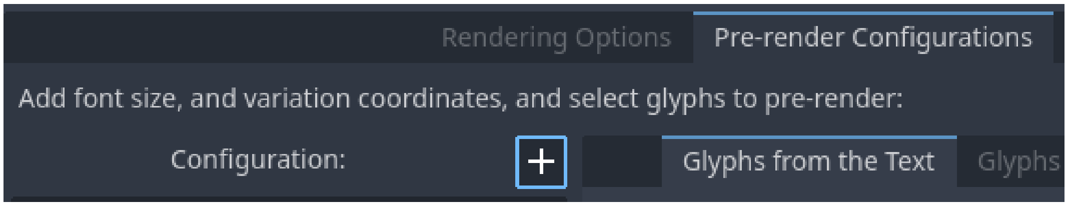
Fallbacks: Array[Font] (size 0)

Compress

Compress: ☑ On

Import dock

Move to the **Pre-render Configurations** tab of the Advanced Import Settings dialog, then add a configuration by clicking the "plus" symbol:


Rendering Options
Pre-render Configurations
Configuration:
+
Glyphs from the Text
Glyphs

Add font size, and variation coordinates, and select glyphs to pre-render:

Adding a new prerendering configuration in the Advanced Import Settings dialog

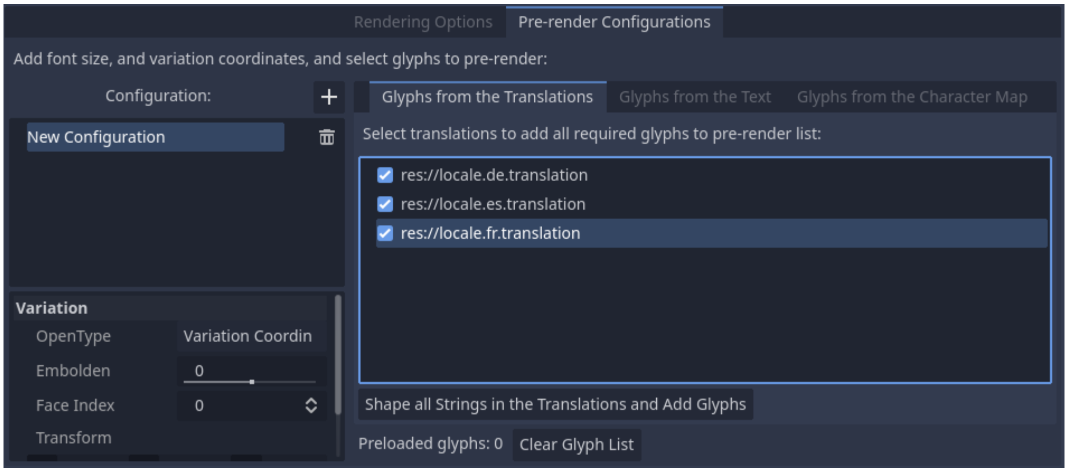
After adding a configuration, make sure it is selected by clicking its name once. You can also rename the configuration by double-clicking it.

There are 2 ways to add glyphs to be prerendered to a given configuration. It is possible to use both approaches in a cumulative manner:

Using text from translations

For most projects, this approach is the most convenient to use, as it automatically sources text from your language translations. The downside is that it can only be used if your project supports internationalization. Otherwise, stick to the "Using custom text" approach described below.

After adding translations to the Project Settings, use the **Glyphs from the Translations** tab to check translations by double-clicking them, then click **Shape All Strings in the Translations and Add Glyphs** at the bottom:


Rendering Options
Pre-render Configurations
Configuration:
+
Glyphs from the Translations
Glyphs from the Text
Glyphs from the Character Map
Shape all Strings in the Translations and Add Glyphs
Clear Glyph List

Add font size, and variation coordinates, and select glyphs to pre-render:

New Configuration

Select translations to add all required glyphs to pre-render list:

☑ res://locale.de.translation
☑ res://locale.es.translation
☑ res://locale.fr.translation

Variation

OpenType: Variation Coordin
Embolden: 0
Face Index: 0
Transform

Preloaded glyphs: 0

Enabling prerendering in the Advanced Import Settings dialog with the **Glyphs from the Translations** tab

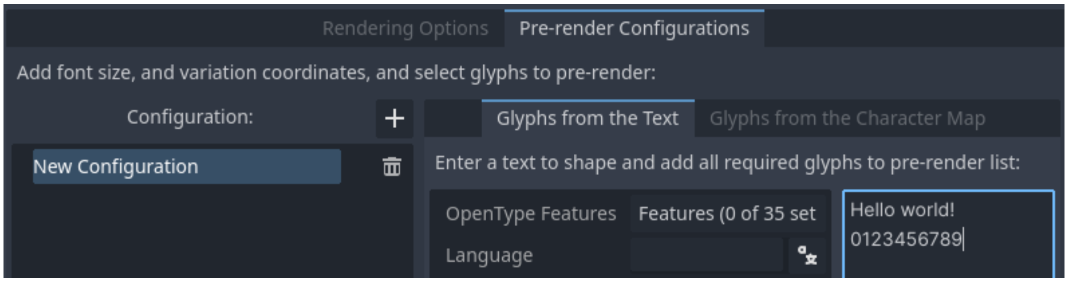
Note

The list of prerendered glyphs is not automatically updated when translations are updated, so you need to repeat this process if your translations have changed significantly.

Using custom text

While it requires manually specifying text that will appear in the game, this is the most efficient approach for games which don't feature user text input. This approach is worth exploring for mobile games to reduce the file size of the distributed app.

To use existing text as a baseline for prerendering, go to the **Glyphs from the Text** sub-tab of the Advanced Import Settings dialog, enter text in the window on the right, then click **Shape Text and Add Glyphs** at the bottom of the dialog:


Rendering Options
Pre-render Configurations
Configuration:
Glyphs from the Text
Glyphs from the Character Map
OpenType Features
Language

Add font size, and variation coordinates, and select glyphs to pre-render:

New Configuration

Enter a text to shape and add all required glyphs to pre-render list:

OpenType Features: Features (0 of 35 set

Hello world!
0123456789

Enabling prerendering in the Advanced Import Settings dialog with the **Glyphs from the Text** tab

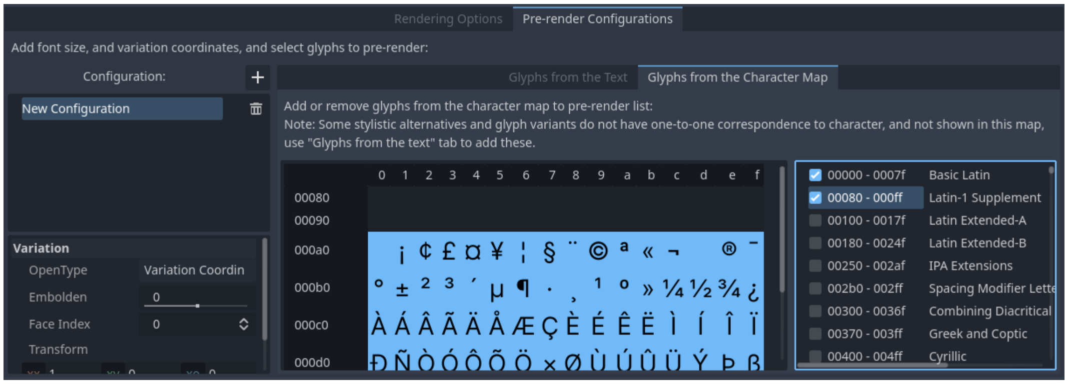
Tip

If your project supports internationalization, you can paste the contents of your CSV or PO files in the above box to quickly prerender all possible characters that may be rendered during gameplay (excluding user-provided or non-translatable strings).

By enabling character sets

The second method requires less configuration and fewer updates if your game's text changes, and is more suited to text-heavy games or multiplayer games with chat. On the other hand, it may cause glyphs that never show up in the game to be prerendered, which is less efficient in terms of file size.

To use existing text as a baseline for prerendering, go to the **Glyphs from the Character Map** sub-tab of the Advanced Import Settings dialog, then *double-click* character sets to be enabled on the right:


Rendering Options
Pre-render Configurations
Configuration:
New Configuration
Glyphs from the Text
Glyphs from the Character Map
Variation
OpenType
Variation Coordin
Transform
0 1 2 3 4 5 6 7 8 9 a b c d e f
00080
00090
000a0
000b0
000c0
000d0

Add font size, and variation coordinates, and select glyphs to pre-render:

Add or remove glyphs from the character map to pre-render list:
Note: Some stylistic alternatives and glyph variants do not have one-to-one correspondence to character, and not shown in this map, use "Glyphs from the text" tab to add these.

Embolden 0

Face Index 0

☑ 00000 - 0007f Basic Latin
☑ 00080 - 000ff Latin-1 Supplement
☐ 00100 - 0017f Latin Extended-A
☐ 00180 - 0024f Latin Extended-B
☐ 00250 - 002af IPA Extensions
☐ 002b0 - 002ff Spacing Modifier Lette
☐ 00300 - 0036f Combining Diacritical
☐ 00370 - 003ff Greek and Coptic
☐ 00400 - 004ff Cyrillic

Enabling prerendering in the Advanced Import Settings dialog with the **Glyphs from the Character Map** tab

To ensure full prerendering, the character sets you need to enable depend on which languages are supported in your game. For English, only **Basic Latin** needs to be enabled. Enabling **Latin-1 Supplement** as well allows fully covering many more languages, such as French, German and Spanish. For Russian, **Cyrillic** needs to be enabled, and so on.

Default project font properties

In the **GUI > Theme** section of the advanced Project Settings, you can choose how the default font should be rendered:

- **Default Font Antialiasing:** Controls the antialiasing method used for the default project font.

- **Default Font Hinting:** Controls the hinting method used for the default project font.
- **Default Font Subpixel Positioning:** Controls the subpixel positioning method for the default project font.
- **Default Font Multichannel Signed Distance Field:** If `true`, makes the default project font use MSDF font rendering instead of traditional rasterization.
- **Default Font Generate Mipmaps:** If `true`, enables mipmap generation and usage for the default project font.

Note

These project settings *only* affect the default project font (the one that is hardcoded in the engine binary).

Custom fonts' properties are governed by their respective import options instead. You can use the **Import Defaults** section of the Project Settings dialog to override default import options for custom fonts.

BBCode in RichTextLabel

Introduction

Label nodes are great for displaying basic text, but they have limitations. If you want to change the color of the text, or its alignment, you can only do that to the entire label. You can't make a part of the text have another color, or have a part of the text centered. To get around these limitations, you would use a RichTextLabel.

RichTextLabel allows for complex formatting of text using a markup syntax or the built-in API. It uses BBCodes for the markup syntax, a system of tags that designate formatting rules for a part of the text. You may be familiar with them if you ever used forums (also known as *bulletin boards*, hence the "BB" in "BBCode").

Unlike Label, RichTextLabel also comes with its own vertical scrollbar. This scrollbar is automatically displayed if the text does not fit within the control's size. The scrollbar can be disabled by unchecking the **Scroll Active** property in the RichTextLabel inspector.

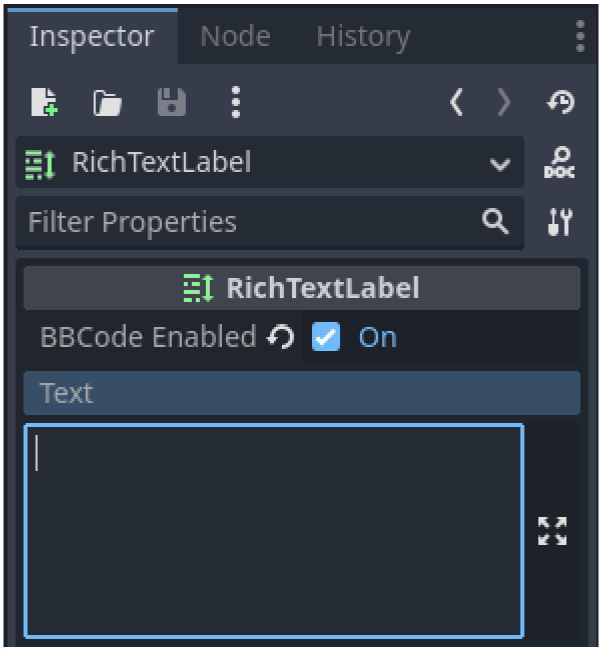
Note that the BBCode tags can also be used to some extent in the XML source of the class reference. For more information, see Class reference primer.

See also

You can see how BBCode in RichTextLabel works in action using the Rich Text Label with BBCode demo project [https://github.com/godotengine/godot-demo-projects/tree/master/gui/rich_text_bbcode].

Using BBCode

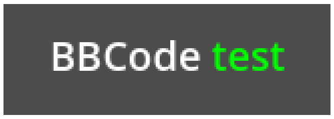
By default, RichTextLabel functions like a normal Label. It has the property_text property, which you can edit to have uniformly formatted text. To be able to use BBCode for rich text formatting, you need to turn on the BBCode mode by setting bbcode_enabled. After that, you can edit the text property using available tags. Both properties are located at the top of the inspector after selecting a RichTextLabel node.



BBCode Enabled ☑ On

Text

For example, `BBCode [color=green]test[/color]` would render the word "test" with a green color.



Most BBCodes consist of 3 parts: the opening tag, the content and the closing tag. The opening tag delimits the start of the formatted part, and can also carry some configuration options. Some opening tags, like the `color` one shown above, also require a value to work.

Other opening tags may accept multiple options (separated by spaces within the opening tag). The closing tag delimits the end of the formatted part. In some cases, both the closing tag and the content can be omitted.

Unlike BBCode in HTML, leading/trailing whitespace is not removed by a RichTextLabel upon display. Duplicate spaces are also displayed as-is in the final output. This means that when displaying a code block in a RichTextLabel, you don't need to use a preformatted text tag.

```
[tag]content[/tag]
[tag=value]content[/tag]
[tag option1=value1 option2=value2]content[/tag]
[tag][/tag]
[tag]
```

Note

RichTextLabel doesn't support entangled BBCode tags. For example, instead of using:

```
[b]bold[i]bold italic[/b]italic[/i]
```

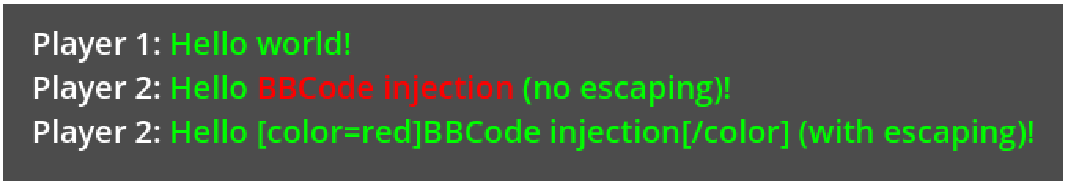
Use:

```
[b]bold[i]bold italic[/i][/b][i]italic[/i]
```

Handling user input safely

In a scenario where users may freely input text (such as chat in a multiplayer game), you should make sure users cannot use arbitrary BBCode tags that will be parsed by RichTextLabel. This is to avoid inappropriate use of formatting, which can be problematic if `[url]` tags are handled by your RichTextLabel (as players may be able to create clickable links to phishing sites or similar).

Using RichTextLabel's `[lb]` and/or `[rb]` tags, we can replace the opening and/or closing brackets of any BBCode tag in a message with those escaped tags. This prevents users from using BBCode that will be parsed as tags – instead, the BBCode will be displayed as text.



Player 1: Hello world!
Player 2: Hello BBCode injection (no escaping)!
Player 2: Hello [color=red]BBCode injection[/color] (with escaping)!

Example of unescaped user input resulting in BBCode injection (2nd line) and escaped user input (3rd line)

The above image was created using the following script:

```
extends RichTextLabel

func _ready():
    append_chat_line("Player 1", "Hello world!")
    append_chat_line("Player 2", "Hello [color=red]BBCode injection[/color] (no escaping)!")
    append_chat_line_escaped("Player 2", "Hello [color=red]BBCode injection[/color] (with escaping)!")


# Returns escaped BBCode that won't be parsed by RichTextLabel as tags.
func escape_bbcode(bbcode_text):
    # We only need to replace opening brackets to prevent tags from being parsed.
    return bbcode_text.replace("[", "[lb]")


# Appends the user's message as-is, without escaping. This is dangerous!
func append_chat_line(username, message):
    append_text("%s: [color=green]%s[/color]\n" % [username, message])
```

```
# Appends the user's message with escaping.
# Remember to escape both the player name and message contents.
func append_chat_line_escaped(username, message):
    append_text("%s: [color=green]%s[/color]\n" %
[escape_bbcode(username), escape_bbcode(message)])
```

Stripping BBCode tags

For certain use cases, it can be desired to remove BBCode tags from the string. This is useful when displaying the RichTextLabel's text in another Control that does not support BBCode (such as a tooltip):

```
extends RichTextLabel

func _ready():
    var regex = RegEx.new()
    regex.compile("\\[.*?\\]")
    var text_without_tags = regex.sub(text, "", true)
    # `text_without_tags` contains the text with all BBCode
tags removed.
```

Note

Removing BBCode tags entirely isn't advised for user input, as it can modify the displayed text without users understanding why part of their message was removed. Escaping user input should be preferred instead.

Performance

In most cases, you can use BBCode directly as-is since text formatting is rarely a heavy task. However, with particularly large RichTextLabels (such as console logs spanning thousands of lines), you may encounter stuttering during gameplay when the RichTextLabel's text is updated.

There are several ways to alleviate this:

- Use the `append_text()` function instead of appending to the `text` property. This function will only parse BBCode for the added text, rather than parsing BBCode from the entire `text` property.
- Use `push_[tag]()` and `pop()` functions to add tags to RichTextLabel instead of using BBCode.
- Enable the **Threading > Threaded** property in RichTextLabel. This won't speed up processing, but it will prevent the main thread from blocking, which avoids stuttering during gameplay. Only enable threading if it's actually needed in your project, as threading has some overhead.

Using push_[tag]() and pop() functions instead of BBCode

If you don't want to use BBCode for performance reasons, you can use functions provided by RichTextLabel to create formatting tags without writing BBCode in the text.

Every BBCode tag (including effects) has a `push_[tag]()` function (where `[tag]` is the tag's name). There are also a few convenience functions available, such as `push_bold_italics()` that combines both `push_bold()` and `push_italics()` into a single tag. See the RichTextLabel class reference for a complete list of `push_[tag]()` functions.

The `pop()` function is used to end *any* tag. Since BBCode is a tag *stack*, using `pop()` will close the most recently started tags first.

The following script will result in the same visual output as using BBCode `[color=green]test [i]example[/i][/color]`:

```
extends RichTextLabel

func _ready():
    append_text("BBCode ")  # Trailing space separates words from each other.
    push_color(Color.GREEN)
    append_text("test ")  # Trailing space separates words from each other.
    push_italics()
    append_text("example")
    pop()  # Ends the tag opened by `push_italics()`.
    pop()  # Ends the tag opened by `push_color()`.
```

Warning

Do **not** set the `text` property directly when using formatting functions. Appending to the `text` property will erase all modifications made to the RichTextLabel using the `append_text()`, `push_[tag]()` and `pop()` functions.

Reference

See also

Some of these BBCode tags can be used in tooltips for `@export` script variables as well as in the XML source of the class reference. For more information, see Class reference BBCode.

Tag	Example

b Makes `{text}` use the bold (or bold italics) font of `RichTextLabel`.	`[b]{text}[/b]`
i Makes `{text}` use the italics (or bold italics) font of `RichTextLabel`.	`[i]{text}[/i]`
u Makes `{text}` underlined.	`[u]{text}[/u]`
s Makes `{text}` strikethrough.	`[s]{text}[/s]`
code Makes `{text}` use the mono font of `RichTextLabel`.	`[code]{text}[/code]`
char Adds Unicode character with hexadecimal UTF-32 `{codepoint}`.	`[char={codepoint}]`
p Adds new paragraph with `{text}`. Supports configuration options, see Paragraph options.	`[p]{text}[/p]` `[p {options}]{text}[/p]`
center Makes `{text}` horizontally centered.	`[center]{text}[/center]`

Same as `[p align=center]`.	
left Makes `{text}` horizontally left-aligned. Same as `[p align=left]`.	`[left]{text}[/left]`
right Makes `{text}` horizontally right-aligned. Same as `[p align=right]`.	`[right]{text}[/right]`
fill Makes `{text}` fill the full width of `RichTextLabel`. Same as `[p align=fill]`.	`[fill]{text}[/fill]`
indent Indents `{text}` once. The indentation width is the same as with `[ul]` or `[ol]`, but without a bullet point.	`[indent]{text}[/indent]`
url Creates a hyperlink (underlined and clickable text). Can contain optional `{text}` or display `{link}` as is. **Must be handled with the "meta_clicked" signal to have an effect,** see [Handling [url] tag clicks]().	`[url]{link}[/url]` `[url={link}]{text}[/url]`
hint	`[hint={tooltip text displayed on hover}]`

Creates a tooltip hint that is displayed when hovering the text with the mouse. Tooltip text should not be quoted (quotes will appear as-is in the tooltip otherwise).	`{text}[/hint]`

img
Inserts an image from the `{path}` (can be any valid Texture2D resource).
If `{width}` is provided, the image will try to fit that width maintaining the aspect ratio.
If both `{width}` and `{height}` are provided, the image will be scaled to that size.
Add `%` to the end of `{width}` or `{height}` value to specify it as percentages of the control width instead of pixels.
If `{valign}` configuration is provided, the image will try to align to the surrounding text, see Image and table vertical alignment.
Supports configuration options, see Image options.
	`[img]{path}[/img]`
	`[img={width}]{path}[/img]`
	`[img={width}x{height}]{path}[/img]`
	`[img={valign}]{path}[/img]`
	`[img {options}]{path}[/img]`

font Makes `{text}` use a font resource from the `{path}`. Supports configuration options, see Font options.	`[font={path}]{text}[/font]` `[font {options}]{text}[/font]`
font_size Use custom font size for `{text}`.	`[font_size={size}]{text}[/font_size]`

dropcap Use a different font size and color for `{text}`, while making the tag's contents span multiple lines if it's large enough. A drop cap is typically one uppercase character, but `[dropcap]` supports containing multiple characters. `margins` values are comma-separated and can be positive, zero or negative. Negative top and bottom margins are particularly useful to allow the rest of the paragraph to display below the dropcap.	`[dropcap font={font} font_size={size} color={color} outline_size={size} outline_color={color} margins={left},{top},{right},{bottom}]{text}[/dropcap]`
opentype_features Enables custom OpenType font features for `{text}`. Features must be provided as a comma-separated `{list}`.	`[opentype_features={list}]{text}[/opentype_features]`
lang Overrides the language for `{text}` that is set by the **BiDi > Language** property in RichTextLabel. `{code}` must be an ISO language code. This can be used to enforce the use of a specific script for a language without starting a new paragraph. Some font files may contain script-specific substitutes, in which case they will be used.	`[lang={code}]{text}[/lang]`

color Changes the color of `{text}`. Color must be provided by a common name (see Named colors) or using the HEX format (e.g. `#ff00ff`, see Hexadecimal color codes).	`[color={code/name}]{text}[/color]`
bgcolor Draws the color behind `{text}`. This can be used to highlight text. Accepts same values as the `color` tag.	`[bgcolor={code/name}]{text}[/bgcolor]`
fgcolor Draws the color in front of `{text}`. This can be used to "redact" text by using an opaque foreground color. Accepts same values as the `color` tag.	`[fgcolor={code/name}]{text}[/fgcolor]`
outline_size Use custom font outline size for `{text}`.	`[outline_size={size}]{text}[/outline_size]`
outline_color Use custom outline color for `{text}`. Accepts same values as the `color` tag.	`[outline_color={code/name}]{text}[/outline_color]`
table Creates a table with the `{number}` of columns. Use the `cell` tag to define	`[table={number}]{cells}[/table]`

table cells. If `{valign}` configuration is provided, the table will try to align to the surrounding text, see Image and table vertical alignment. If baseline alignment is used, the table is aligned to the baseline of the row with index `{alignment_row}` (zero-based).	`[table={number},{valign}]{cells}[/table]` `[table={number},{valign},{alignment_row}]{cells}[/table]`

cell
Adds a cell with `{text}` to the table.
If `{ratio}` is provided, the cell will try to expand to that value proportionally to other cells and their ratio values.
Supports configuration options, see Cell options.
	`[cell]{text}[/cell]`
	`[cell={ratio}]{text}[/cell]`
	`[cell {options}]{text}[/cell]`

ul Adds an unordered list. List `{items}` must be provided by putting one item per line of text. The bullet point can be customized using the `{bullet}` parameter, see Unordered list bullet.	`[ul]{items}[/ul]` `[ul bullet={bullet}]{items}[/ul]`
ol Adds an ordered (numbered) list of the given `{type}` (see Ordered list types). List `{items}` must be provided by putting one item per line of text.	`[ol type={type}]{items}[/ol]`

lb, **rb** Adds `[` and `]` respectively. Allows escaping BBCode markup. These are self-closing tags, which means you do not need to close them (and there is no `[/lb]` or `[/rb]` closing tag).	`[lb]b[rb]text[lb]/b[rb]` will display as `[b]text[/b]`.
Several Unicode control characters can be added using their own self-closing tags. This can result in easier maintenance compared to pasting those control characters directly in the text.	`[lrm]` (left-to-right mark), `[rlm]` (right-to-left mark), `[lre]` (left-to-right embedding), `[rle]` (right-to-left embedding), `[lro]` (left-to-right override), `[rlo]` (right-to-left override), `[pdf]` (pop directional formatting), `[alm]` (Arabic letter mark), `[lri]` (left-to-right isolate), `[rli]` (right-to-left isolate), `[fsi]` (first strong isolate), `[pdi]` (pop directional isolate), `[zwj]` (zero-width joiner), `[zwnj]` (zero-width non-joiner), `[wj]` (word joiner), `[shy]` (soft hyphen)

Note

Tags for bold (`[b]`) and italics (`[i]`) formatting work best if the appropriate custom fonts are set up in the RichTextLabelNode's theme overrides. If no custom bold or italic fonts are defined, faux bold and italic fonts [https://fonts.google.com/knowledge/glossary/faux_fake_pseudo_synthesized] will be generated by Godot. These fonts rarely look good in comparison to hand-made bold/italic font variants.

The monospaced (`[code]`) tag **only** works if a custom font is set up in the RichTextLabel node's theme overrides. Otherwise, monospaced text will use the regular font.

There are no BBCode tags to control vertical centering of text yet.

Options can be skipped for all tags.

Paragraph options

- **align**

Values	`left` (or `l`), `center` (or `c`), `right` (or `r`), `fill` (or `f`)
Default	`left`

Text horizontal alignment.

- **bidi_override**, **st**

Values	`default` (of `d`), `uri` (or `u`), `file` (or `f`), `email` (or `e`), `list` (or `l`), `none` (or `n`), `custom` (or `c`)

Default	`default`

Structured text override.

- **justification_flags**, **jst**

Values	Comma-separated list of the following values: `kashida` (or `k`), `word` (or `w`), `trim` (or `tr`), `after_last_tab` (or `lt`), `skip_last` (or `sl`), `skip_last_with_chars` (or `sv`), `do_not_skip_single` (or `ns`).
Default	`word,kashida,skip_last,do_not_skip_single`

Justification (fill alignment) option. See TextServer for more details.

- **direction**, **dir**

Values	`ltr` (or `l`), `rtl` (or `r`), `auto` (or `a`)
Default	Inherit

Base BiDi direction.

- **language**, **lang**

Values	ISO language codes. See Locale codes

Default	Inherit

Locale override. Some font files may contain script-specific substitutes, in which case they will be used.

- **tab_stops**

Values	List of floating-point numbers, e.g. `10.0,30.0`
Default	Width of the space character in the font

Overrides the horizontal offsets for each tab character. When the end of the list is reached, the tab stops will loop over. For example, if you set `tab_stops` to `10.0,30.0`, the first tab will be at `10` pixels, the second tab will be at `10 + 30 = 40` pixels, and the third tab will be at `10 + 30 + 10 = 50` pixels from the origin of the RichTextLabel.

Handling `[url]` tag clicks

By default, `[url]` tags do nothing when clicked. This is to allow flexible use of `[url]` tags rather than limiting them to opening URLs in a web browser.

To handle clicked `[url]` tags, connect the `RichTextLabel` node's meta_clicked signal to a script function.

For example, the following method can be connected to `meta_clicked` to open clicked URLs using the user's default web browser:

```
# This assumes RichTextLabel's `meta_clicked` signal was connected to
# the function below using the signal connection dialog.
func _richtextlabel_on_meta_clicked(meta):
    # `meta` is not guaranteed to be a String, so convert it to a String
    # to avoid script errors at run-time.
    OS.shell_open(str(meta))
```

For more advanced use cases, it's also possible to store JSON in an `[url]` tag's option and parse it in the function that handles the `meta_clicked` signal. For example:

```
[url={"example": "value"}]JSON[/url]
```

Image options

- **color**

Values	Color name or color in HEX format
Default	Inherit

  Color tint of the image (modulation).

- **height**

Values	Integer number
Default	Inherit

Target height of the image in pixels, add `%` to the end of value to specify it as percentages of the control width instead of pixels.

- **width**

Values	Integer number
Default	Inherit

Target width of the image, add `%` to the end of value to specify it as percentages of the control width instead of pixels.

- **region**

Values	x,y,width,height in pixels
Default	Inherit

Region rect of the image. This can be used to display a single image from a spritesheet.

- **pad**

Values	`false`, `true`
Default	`false`

If set to `true`, and the image is smaller than the size specified by `width` and `height`, the image padding is added to match the size instead of upscaling.

- **tooltip**

Values	String
Default	

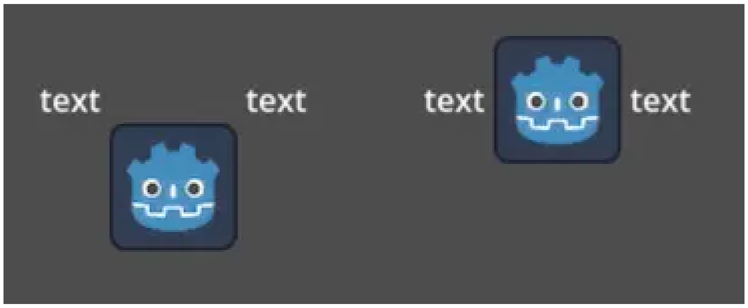
Image tooltip.

Image and table vertical alignment

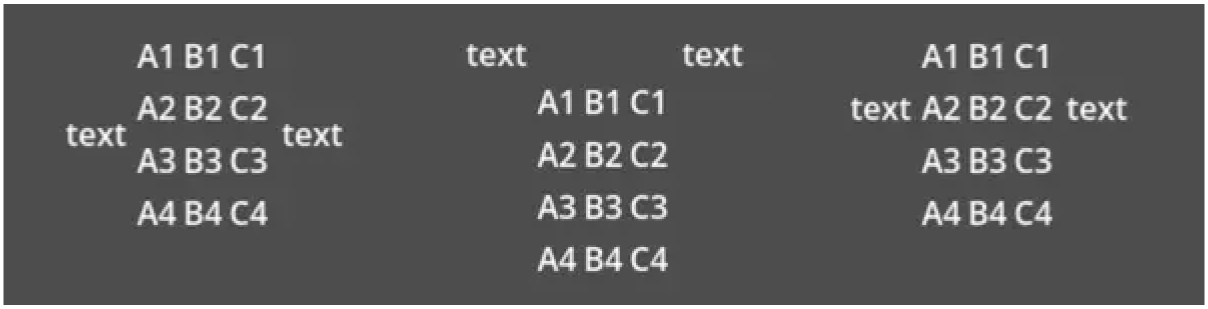
When a vertical alignment value is provided with the `[img]` or `[table]` tag the image/table will try to align itself against the surrounding text. Alignment is performed using a vertical point of the image and a vertical point of the text. There are 3 possible points on the image (`top`, `center`, and `bottom`) and 4 possible points on the text and table (`top`, `center`, `baseline`, and `bottom`), which can be used in any combination.

To specify both points, use their full or short names as a value of the image/table tag:

```
text [img=top,bottom]...[/img] text
text [img=center,center]...[/img] text
```



```
text [table=3,center]...[/table] text  # Center to center.
text [table=3,top,bottom]...[/table] text # Top of the table to the bottom of text.
text [table=3,baseline,baseline,1]...[/table] text # Baseline of the second row (rows use zero-based indexing) to the baseline of text.
```


A1 B1 C1
A2 B2 C2
text
text
A3 B3 C3
A4 B4 C4
text
text
A1 B1 C1
A2 B2 C2
A3 B3 C3
A4 B4 C4
A1 B1 C1
text A2 B2 C2 text
A3 B3 C3
A4 B4 C4

You can also specify just one value (`top`, `center`, or `bottom`) to make use of a corresponding preset (`top-top`, `center-center`, and `bottom-bottom` respectively).

Short names for the values are `t` (`top`), `c` (`center`), `l` (`baseline`), and `b` (`bottom`).

Font options

- **name**, **n**

Values	A valid Font resource path.
Default	Inherit

Font resource path.

- **size**, **s**

Values	Number in pixels.
Default	Inherit

Custom font size.

- **glyph_spacing**, **gl**

Values	Number in pixels.
Default	Inherit

Extra spacing for each glyph.

- **space_spacing**, **sp**

Values	Number in pixels.
Default	Inherit

Extra spacing for the space character.

- **top_spacing**, **top**

Values	Number in pixels.
Default	Inherit

Extra spacing at the top of the line.

- **bottom_spacing**, **bt**

Values	Number in pixels.
Default	Inherit

  Extra spacing at the bottom of the line.

- **embolden**, **emb**

Values	Floating-point number.
Default	`0.0`

  Font embolden strength, if it is not equal to zero, emboldens the font outlines. Negative values reduce the outline thickness.

- **face_index**, **fi**

Values	Integer number.
Default	`0`

  An active face index in the TrueType / OpenType collection.

- **slant**, **sln**

Values	Floating-point number.
Default	`0.0`

Font slant strength, positive values slant glyphs to the right. Negative values to the left.

- **opentype_variation**, **otv**

Values	Comma-separated list of the OpenType variation tags.
Default	

Font OpenType variation coordinates. See [OpenType variation tags](https://docs.microsoft.com/en-us/typography/opentype/spec/dvaraxisreg) [https://docs.microsoft.com/en-us/typography/opentype/spec/dvaraxisreg].

Note: The value should be enclosed in `"` to allow using `=` inside it:

```
[font otv="wght=200,wdth=400"] # Sets variable font weight and
width.
```

- **opentype_features**, **otf**

Values	Comma-separated list of the OpenType feature tags.
Default	

Font OpenType features. See [OpenType features tags](https://docs.microsoft.com/en-us/typography/opentype/spec/featuretags) [https://docs.microsoft.com/en-us/typography/opentype/spec/featuretags].

Note: The value should be enclosed in `"` to allow using `=` inside it:

```
[font otf="calt=0,zero=1"] # Disable contextual alternates,
enable slashed zero.
```

Named colors

For tags that allow specifying a color by name you can use names of the constants from the built-in Color class. Named classes can be specified in a number of styles using different casings: `DARK_RED`, `DarkRed`, and `darkred` will give the same exact result.

Hexadecimal color codes

For opaque RGB colors, any valid 6-digit hexadecimal code is supported, e.g. `[color=#ffffff]white[/color]`. Shorthand RGB color codes such as `#6f2` (equivalent to `#66ff22`) are also supported.

For transparent RGB colors, any RGBA 8-digit hexadecimal code can be used, e.g. `[color=#ffffff88]translucent white[/color]`. Note that the alpha channel is the **last** component of the color code, not the first one. Short RGBA color codes such as `#6f28` (equivalent to `#66ff2288`) are supported as well.

Cell options

- **expand**

Values	Integer number

Default	1

Cell expansion ratio. This defines which cells will try to expand to proportionally to other cells and their expansion ratios.

- **border**

Values	Color name or color in HEX format
Default	Inherit

  Cell border color.

- **bg**

Values	Color name or color in HEX format
Default	Inherit

  Cell background color. For alternating odd/even row backgrounds, you can use `bg=odd_color,even_color`.

- **padding**

Values	4 comma-separated floating-point numbers
Default	0, 0, 0, 0

Left, top, right, and bottom cell padding.

Unordered list bullet

By default, the `[ul]` tag uses the `U+2022` "Bullet" Unicode glyph as the bullet character. This behavior is similar to web browsers. The bullet character can be customized using `[ul bullet={bullet}]`. If provided, this `{bullet}` parameter must be a *single* character with no enclosing quotes (for example, `[bullet=*]`). Additional characters are ignored. The bullet character's width does not affect the list's formatting.

See Bullet (typography) on Wikipedia [https://en.wikipedia.org/wiki/Bullet_(typography)] for a list of common bullet characters that you can paste directly in the `bullet` parameter.

Ordered list types

Ordered lists can be used to automatically mark items with numbers or letters in ascending order. This tag supports the following type options:

- `1` - Numbers, using language specific numbering system if possible.
- `a`, `A` - Lower and upper case Latin letters.
- `i`, `I` - Lower and upper case Roman numerals.

Text effects

BBCode can also be used to create different text effects that can optionally be animated. Five customizable effects are provided out of the box, and you can easily create your own. By default, animated effects will pause when the SceneTree is paused. You can change this behavior by adjusting the RichTextLabel's **Process > Mode** property.

All examples below mention the default values for options in the listed tag format.

Note

Text effects that move characters' position may result in characters being clipped by the RichTextLabel node bounds.

You can resolve this by disabling **Control > Layout > Clip Contents** in the inspector after selecting the RichTextLabel node, or ensuring there is enough margin added around the text by using line breaks above and below the line using the effect.

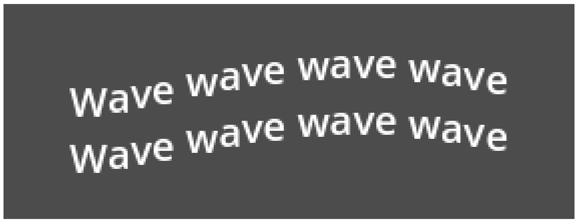
Pulse

Pulse Pulse Pulse Pulse
Pulse Pulse Pulse Pulse

Pulse creates an animated pulsing effect that multiplies each character's opacity and color. It can be used to bring attention to specific text. Its tag format is `[pulse freq=1.0 color=#ffffff40 ease=-2.0]{text}[/pulse]`.

`freq` controls the frequency of the half-pulsing cycle (higher is faster). A full pulsing cycle takes `2 * (1.0 / freq)` seconds. `color` is the target color multiplier for blinking. The default mostly fades out text, but not entirely. `ease` is the easing function exponent to use. Negative values provide in-out easing, which is why the default is `-2.0`.

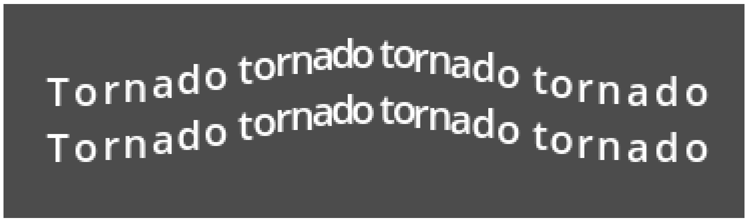
Wave



Wave makes the text go up and down. Its tag format is `[wave amp=50.0 freq=5.0 connected=1]{text}[/wave]`.

`amp` controls how high and low the effect goes, and `freq` controls how fast the text goes up and down. A `freq` value of `0` will result in no visible waves, and negative `freq` values won't display any waves either. If `connected` is `1` (default), glyphs with ligatures will be moved together. If `connected` is `0`, each glyph is moved individually even if they are joined by ligatures. This can work around certain rendering issues with font ligatures.

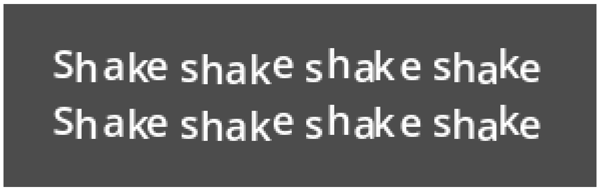
Tornado


Tornado tornado tornado tornado
Tornado tornado tornado tornado

Tornado makes the text move around in a circle. Its tag format is `[tornado radius=10.0 freq=1.0 connected=1]{text}[/tornado]`.

`radius` is the radius of the circle that controls the offset, `freq` is how fast the text moves in a circle. A `freq` value of `0` will pause the animation, while negative `freq` will play the animation backwards. If `connected` is `1` (default), glyphs with ligatures will be moved together. If `connected` is `0`, each glyph is moved individually even if they are joined by ligatures. This can work around certain rendering issues with font ligatures.

Shake

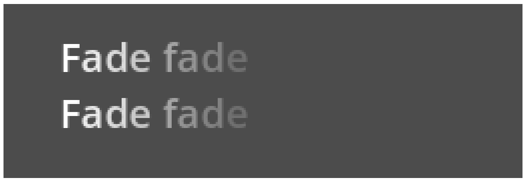


Shake shake shake shake
Shake shake shake shake

Shake makes the text shake. Its tag format is `[shake rate=20.0 level=5 connected=1]{text}[/shake]`.

`rate` controls how fast the text shakes, `level` controls how far the text is offset from the origin. If `connected` is `1` (default), glyphs with ligatures will be moved together. If `connected` is `0`, each glyph is moved individually even if they are joined by ligatures. This can work around certain rendering issues with font ligatures.

Fade

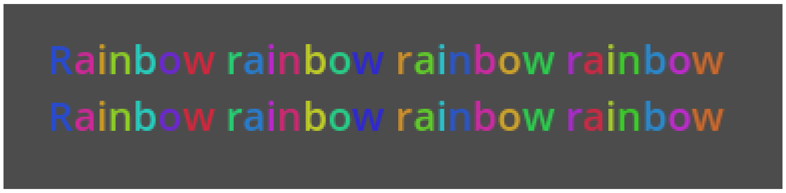


Fade fade
Fade fade

Fade creates a static fade effect that multiplies each character's opacity. Its tag format is `[fade start=4 length=14]{text}[/fade]`.

`start` controls the starting position of the falloff relative to where the fade command is inserted, `length` controls over how many characters should the fade out take place.

Rainbow



Rainbow rainbow rainbow rainbow
Rainbow rainbow rainbow rainbow

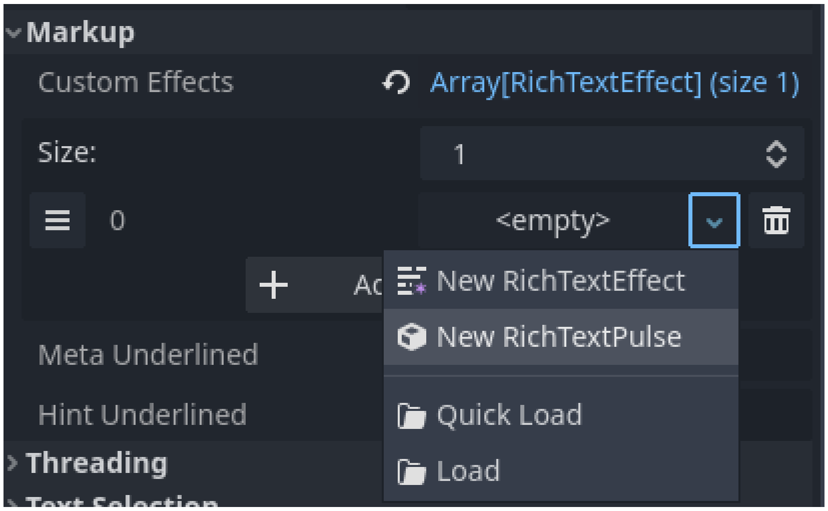
Rainbow gives the text a rainbow color that changes over time. Its tag format is `[rainbow freq=1.0 sat=0.8 val=0.8]{text}[/rainbow]`.

`freq` is the number of full rainbow cycles per second, `sat` is the saturation of the rainbow, `val` is the value of the rainbow. A `freq` value of `0` will pause the animation, while negative `freq` will play the animation backwards.

Font outlines are *not* affected by the rainbow effect (they keep their original color). Existing font colors are overridden by the rainbow effect. However, CanvasItem's **Modulate** and **Self Modulate** properties will affect how the rainbow effect looks, as modulation multiplies its final colors.

Custom BBCode tags and text effects

You can extend the RichTextEffect resource type to create your own custom BBCode tags. Create a new script file that extends the RichTextEffect resource type and give the script a `class_name` so that the effect can be selected in the inspector. Add the `@tool` annotation to your GDScript file if you wish to have these custom effects run within the editor itself. The RichTextLabel does not need to have a script attached, nor does it need to be running in `tool` mode. The new effect can be registered in the Inspector by adding it to the **Markup > Custom Effects** array, or in code with the install_effect() method:



Markup
Custom Effects
Array[RichTextEffect] (size 1)
Size:
1
0
<empty>
Ac
New RichTextEffect
New RichTextPulse
Quick Load
Load
Meta Underlined
Hint Underlined
Threading
Text Selection

Selecting a custom RichTextEffect after saving a script that extends RichTextEffect with a `class_name`

Warning

If the custom effect is not registered within the RichTextLabel's **Markup > Custom Effects** property, no effect will be visible and the original tag will be left as-is.

There is only one function that you need to extend: `_process_custom_fx(char_fx)`. Optionally, you can also provide a custom BBCode identifier simply by adding a member name `bbcode`. The code will check the `bbcode` property automatically or will use the name of the file to determine what the BBCode tag should be.

`_process_custom_fx`

This is where the logic of each effect takes place and is called once per glyph during the draw phase of text rendering. This passes in a CharFXTransform object, which holds a few variables to control how the associated glyph is rendered:

- `identity` specifies which custom effect is being processed. You should use that for code flow control.
- `outline` is `true` if effect is called for drawing text outline.
- `range` tells you how far into a given custom effect block you are in as an index.
- `elapsed_time` is the total amount of time the text effect has been running.
- `visible` will tell you whether the glyph is visible or not and will also allow you to hide a given portion of text.
- `offset` is an offset position relative to where the given glyph should render under normal circumstances.
- `color` is the color of a given glyph.
- `glyph_index` and `font` is glyph being drawn and font data resource used to draw it.
- Finally, `env` is a Dictionary of parameters assigned to a given custom effect. You can use get() with an optional default value to retrieve each parameter, if specified by the user. For example `[custom_fx spread=0.5 color=#FFFF00]test[/custom_fx]` would have a float `spread` and Color `color` parameters in its `env` Dictionary. See below for more usage examples.

The last thing to note about this function is that it is necessary to return a boolean `true` value to verify that the effect processed correctly. This way, if there's a problem with rendering a given glyph, it will back out of rendering custom effects entirely until the user fixes whatever error cropped up in their custom effect logic.

Here are some examples of custom effects:

Ghost

```
@tool
extends RichTextEffect
class_name RichTextGhost

# Syntax: [ghost freq=5.0 span=10.0][/ghost]
# Define the tag name.
var bbcode = "ghost"

func _process_custom_fx(char_fx):
    # Get parameters, or use the provided default value if
missing.
    var speed = char_fx.env.get("freq", 5.0)
    var span = char_fx.env.get("span", 10.0)

    var alpha = sin(char_fx.elapsed_time * speed +
(char_fx.range.x / span)) * 0.5 + 0.5
    char_fx.color.a = alpha
    return true
```

Matrix

```
@tool
extends RichTextEffect
class_name RichTextMatrix

# Syntax: [matrix clean=2.0 dirty=1.0 span=50][/matrix]

# Define the tag name.
var bbcode = "matrix"

# Gets TextServer for retrieving font information.
func get_text_server():
    return TextServerManager.get_primary_interface()

func _process_custom_fx(char_fx):
    # Get parameters, or use the provided default value if
missing.
    var clear_time = char_fx.env.get("clean", 2.0)
    var dirty_time = char_fx.env.get("dirty", 1.0)
    var text_span = char_fx.env.get("span", 50)

    var value = char_fx.glyph_index

    var matrix_time = fmod(char_fx.elapsed_time +
(char_fx.range.x / float(text_span)), \
                           clear_time + dirty_time)

    matrix_time = 0.0 if matrix_time < clear_time else \
                  (matrix_time - clear_time) / dirty_time

    if matrix_time > 0.0:
        value = int(1 * matrix_time * (126 - 65))
        value %= (126 - 65)
        value += 65
    char_fx.glyph_index =
get_text_server().font_get_glyph_index(char_fx.font, 1, value,
0)
    return true
```

This will add a few new BBCode commands, which can be used like so:

```
[center][ghost]This is a custom [matrix]effect[/matrix][/ghost]
made in
[pulse freq=5.0 height=2.0][pulse color=#00FFAA
freq=2.0]GDScript[/pulse][/pulse].[/center]
```

XR

This section of the manual covers everything related to XR (Virtual Reality and Augmented Reality).

Basic Tutorial

- Setting up XR
- Deploying to Android
- A better XR start script
- AR / Passthrough
- Where to go from here

Advanced topics

- OpenXR Settings
- The XR action map
- Room scale in XR
- OpenXR composition layers
- OpenXR hand tracking
- OpenXR body tracking

Godot XR Tools

- Introducing XR tools
- Basic XR Locomotion

Setting up XR

Introduction to the XR system in Godot

Godot provides a modular XR system that abstracts many of the different XR platform specifics away from the user. At the core sits the XRServer which acts as a central interface to the XR system that allows users to discover interfaces and interact with the components of the XR system.

Each supported XR platform is implemented as an XRInterface. Supported interfaces register themselves with the XRServer and can be queried with the `find_interface` method on the XRServer. When the desired interface is found it can be initialized by calling `initialize` on the interface.

Warning

A registered interface means nothing more than that the interface is available, if the interface is not supported by the host system, initialization may fail and return `false`. This can have many reasons and sadly the reasons differ from platform to platform. It can be because the user hasn't installed the required software, or that the user simply hasn't plugged in their headset. You as a developer must thus react properly on an interface failing to initialize.

Due to the special requirements for output in XR, especially for head mounted devices that supply different images to each eye, the XRServer in Godot will override various features in the rendering system. For stand-alone devices this means the final output is handled by the XRInterface and Godot's usual output system is disabled. For desktop XR devices that work as a second screen it is possible to dedicate a separate Viewport to handle the XR output, leaving the main Godot window available for displaying alternative content.

Note

Note that only one interface can be responsible for handling the output to an XR device, this is known as the primary interface and by default will be the first interface that is initialized. Godot currently thus only supports implementations with a single headset. It is possible, but increasingly uncommon, to have a secondary interface, for example to add tracking to an otherwise 3DOF only device.

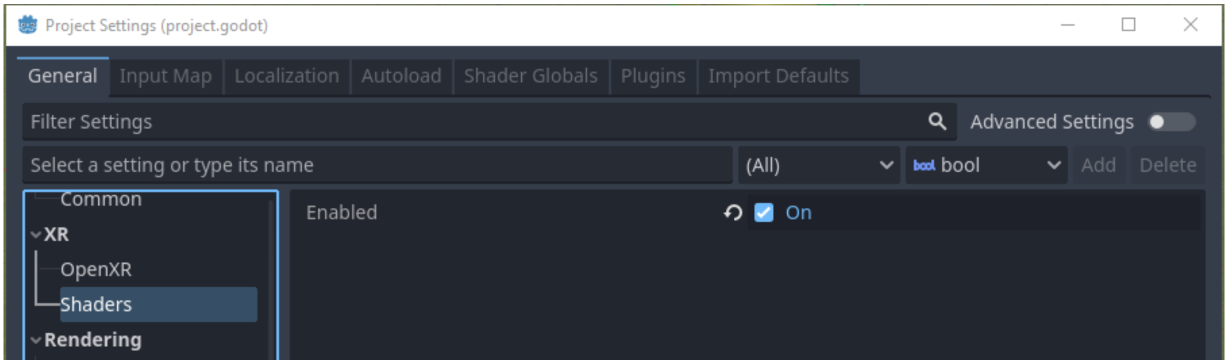
There are three XR specific node types that you will find in nearly all XR applications:

- XROrigin3D represents, for all intents and purposes, the center point of your play space. That is an oversimplified statement but we'll go into more detail later. All objects tracked in physical space by the XR platform are positioned in relation to this point.
- XRCamera3D represents the (stereo) camera that is used when rendering output for the XR device. The positioning of this node is controlled by the XR system and updated automatically using the tracking information provided by the XR platform.
- XRController3D represents a controller used by the player, commonly there are two, one held in each hand. These nodes give access to various states on these controllers and send out signals when the player presses buttons on them. The positioning of this node is controlled by the XR system and updated automatically using the tracking information provided by the XR platform.

There are other XR related nodes and there is much more to say about these three nodes, but we'll get into that later on.

Prerequisites for XR in Godot 4

While in Godot 3 most things worked out of the box, Godot 4 needs a little more setup. This is mainly due to the more advanced nature of the Vulkan renderer. There are many rendering features in Vulkan the XR system uses that aren't enabled by default. They are very easy to turn on, simply open up your project settings and tick the XR shaders tickbox in the XR section:



Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Filter Settings
Advanced Settings
Select a setting or type its name
(All)
bool
Add
Delete
Common
XR
OpenXR
Shaders
Rendering

Enabled ☑ On

Warning

As Godot 4 is still in development, many post process effects have not yet been updated to support stereoscopic rendering. Using these will have adverse effects.

Note

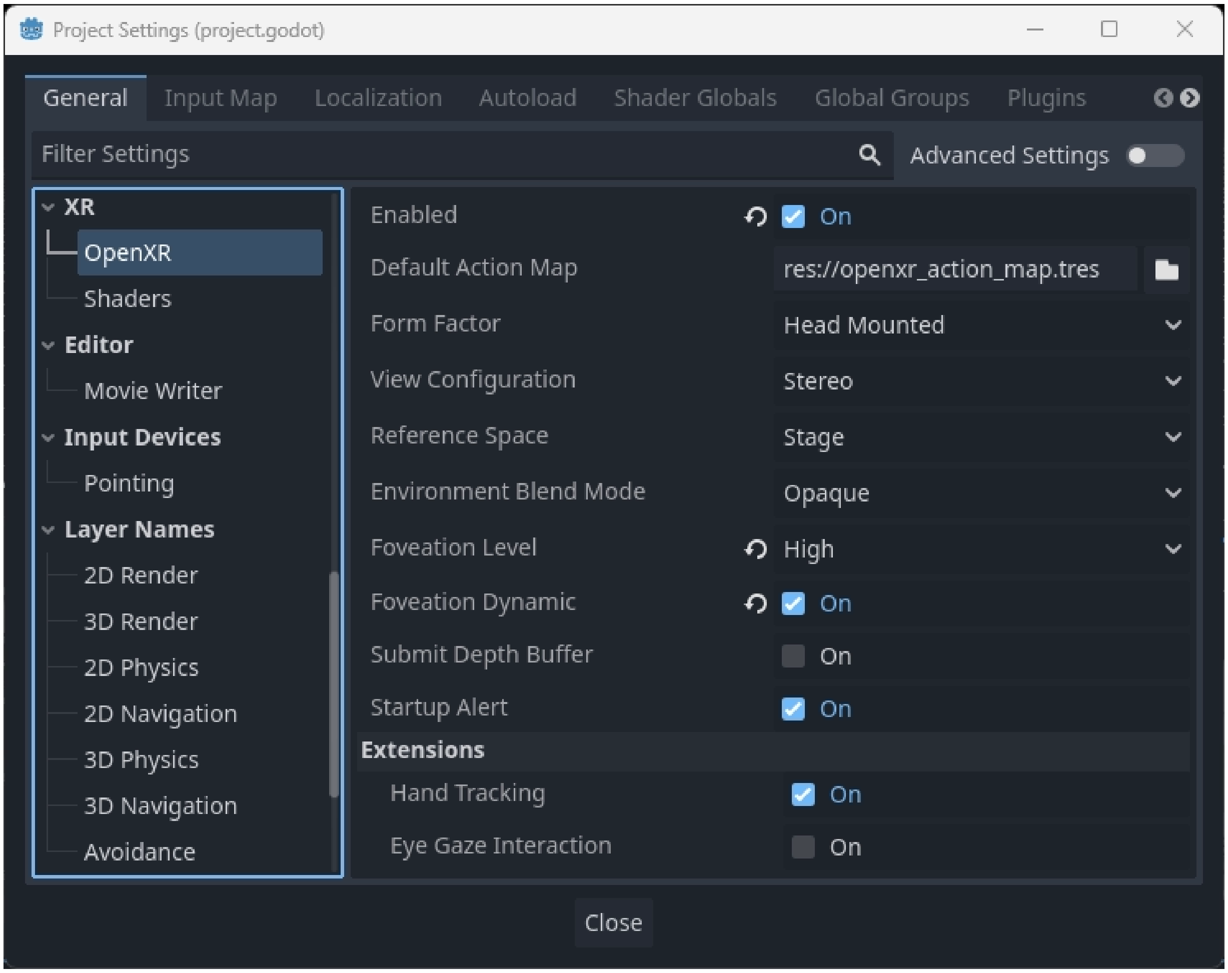
Godot 4 has 3 renderer options, Compatibility, Mobile, and Forward+. In the future XR desktop projects should use Forward+, and projects for stand-alone headsets should use Mobile. However Compatibility is the recommended renderer for now due to it having the best XR performance.

OpenXR

OpenXR is a new industry standard that allows different XR platforms to present themselves through a standardised API to XR applications. This standard is an open standard maintained by the Khronos Group and thus aligns very well with Godot's interests.

The Vulkan implementation of OpenXR is closely integrated with Vulkan, taking over part of the Vulkan system. This requires tight integration of certain core graphics features in the Vulkan renderer which are needed before the XR system is setup. This was one of the main deciding factors to include OpenXR as a core interface.

This also means OpenXR needs to be enabled when Godot starts in order to set things up correctly. The required setting can be found in your project settings:

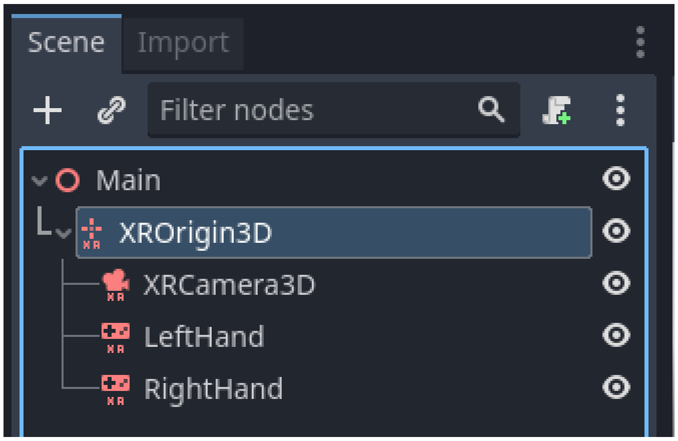

Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Filter Settings
Advanced Settings
XR
OpenXR
Shaders
Editor
Movie Writer
Input Devices
Pointing
Layer Names
2D Render
3D Render
2D Physics
2D Navigation
3D Physics
3D Navigation
Avoidance
Close

Enabled ☑ On
Default Action Map res://openxr_action_map.tres
Form Factor Head Mounted
View Configuration Stereo
Reference Space Stage
Environment Blend Mode Opaque
Foveation Level High
Foveation Dynamic ☑ On
Submit Depth Buffer ☐ On
Startup Alert ☑ On
Extensions
Hand Tracking ☑ On
Eye Gaze Interaction ☐ On

As setup is brought forward with OpenXR you can find several other settings related to OpenXR here as well. These can't be changed while your application is running. The default settings will get us started and we will go into detail in another section of the documentation.

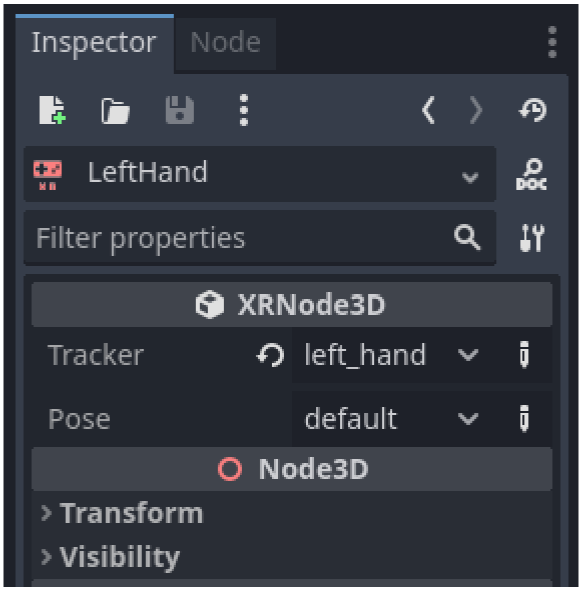
Setting up the XR scene

Every XR application needs at least an XROrigin3D and an XRCamera3D node. Most will have two XRController3D, one for the left hand and one for the right. Keep in mind that the camera and controller nodes should be children of the origin node. Add these nodes to a new scene and rename the controller nodes to `LeftHand` and `RightHand`, your scene should look something like this:


Scene
Import
Filter nodes

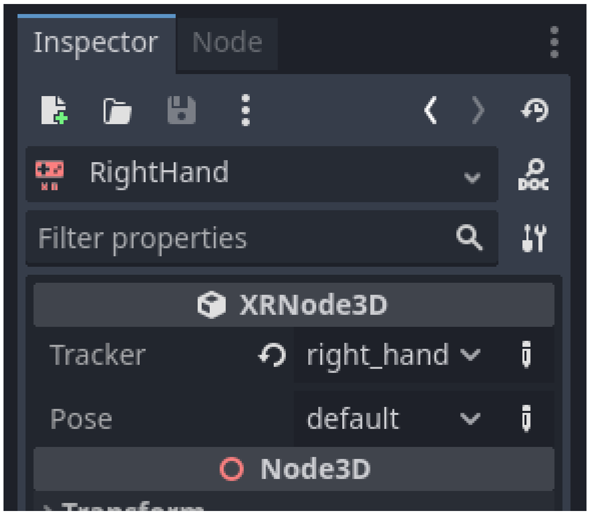
Main
XROrigin3D
XRCamera3D
LeftHand
RightHand

Next you need to configure the controllers, select the left hand and set it up as follows:


Inspector
Node
LeftHand
Filter properties
XRNode3D
Node3D
Transform
Visibility

Tracker left_hand
Pose default

And the right hand:



Tracker: right_hand

Pose: default

Right now all these nodes are on the floor, they will be positioned correctly in runtime. To help during development, it can be helpful to move the camera upwards so its `y` is set to `1.7`, and move the controller nodes to `-0.5, 1.0, -0.5` and `0.5, 1.0, -0.5` for respectively the left and right hand.

Next we need to add a script to our root node. Add the following code into this script:

GDScript

```
extends Node3D

var xr_interface: XRInterface

func _ready():
    xr_interface = XRServer.find_interface("OpenXR")
    if xr_interface and xr_interface.is_initialized():
        print("OpenXR initialized successfully")

        # Turn off v-sync!

DisplayServer.window_set_vsync_mode(DisplayServer.VSYNC_DISABLE
D)

        # Change our main viewport to output to the HMD
        get_viewport().use_xr = true
    else:
        print("OpenXR not initialized, please check if your
headset is connected")
```

C#

```
using Godot;

public partial class MyNode3D : Node3D
{
    private XRInterface _xrInterface;

    public override void _Ready()
    {
        _xrInterface = XRServer.FindInterface("OpenXR");
        if(_xrInterface != null &&
_xrInterface.IsInitialized())
        {
            GD.Print("OpenXR initialized successfully");

            // Turn off v-sync!

DisplayServer.WindowSetVsyncMode(DisplayServer.VSyncMode.Disabl
ed);

            // Change our main viewport to output to the HMD
            GetViewport().UseXR = true;
        }
        else
        {
            GD.Print("OpenXR not initialized, please check if
your headset is connected");
        }
    }
}
```

This code fragment assumes we are using OpenXR, if you wish to use any of the other interfaces you can change the `find_interface` call.

Warning

As you can see in the code snippet above, we turn off v-sync. When using OpenXR you are outputting the rendering results to an HMD that often requires us to run at 90Hz or higher. If your monitor is a 60hz monitor and v-sync is turned on, you will limit the output to 60 frames per second.

XR interfaces like OpenXR perform their own sync.

Also note that by default the physics engine runs at 60Hz as well and this can result in choppy physics. You should set `Engine.physics_ticks_per_second` to a higher value.

If you run your project at this point in time, everything will work but you will be in a dark world. So to finish off our starting point add a DirectionalLight3D and a WorldEnvironment node to your scene. You may wish to also add a mesh instance as a child to each controller node just to temporarily visualise them. Make sure you configure a sky in your world environment.

Now run your project, you should be floating somewhere in space and be able to look around.

Note

While traditional level switching can definitely be used with XR applications, where this scene setup is repeated in each level, most find it easier to set this up once and loading levels as a subscene. If you do switch scenes and replicate the XR setup in each one, do make sure you do not run `initialize` multiple times. The effect can be unpredictable depending on the XR interface used.

For the rest of this basic tutorial series we will create a game that uses a single scene.

Deploying to Android

Setup

Most standalone headsets run on Android and OpenXR support is making its way to these platforms.

Before following the OpenXR-specific instructions here, you'll need to first setup your system to export to Android in general, including:

- Installing OpenJDK 17
- Installing Android Studio
- Creating a debug.keystore
- Configuring the location of the Android SDK and debug.keystore in Godot

See Exporting for Android for the full details, and return here when you've finished these steps.

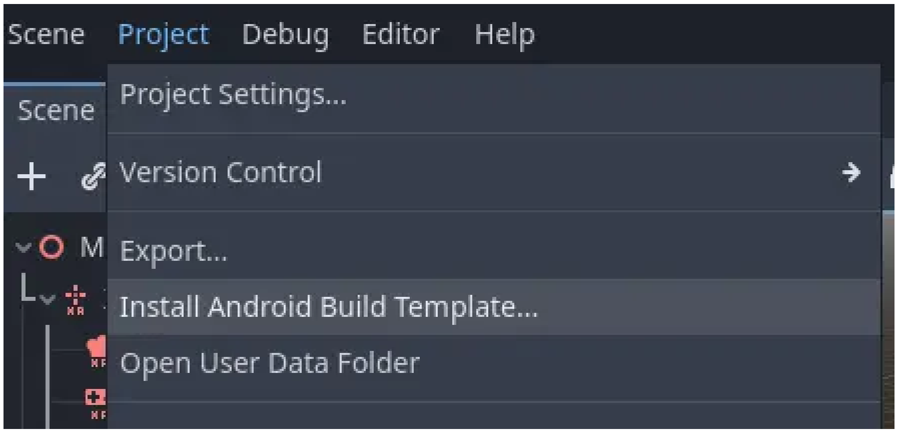
Gradle Android build

Note

Official support for the Android platform wasn't added to the OpenXR specification initially resulting in various vendors creating custom loaders to make OpenXR available on their headsets. While the long term expectation is that all vendors will adopt the official OpenXR loader, for now these loaders need to be added to your project.

In order to include the vendor-specific OpenXR loader into your project, you will need to setup a gradle Android build.

Select **Install Android Build Template...** from the **Project** menu:


Scene
Project
Debug
Editor
Help
Scene
Project Settings...
Version Control
Export...
Install Android Build Template...
Open User Data Folder

This will create a folder called **android** inside of your project that contains all the runtime files needed on Android. You can now customize this installation. Godot won't show this in the editor but you can find it with a file browser.

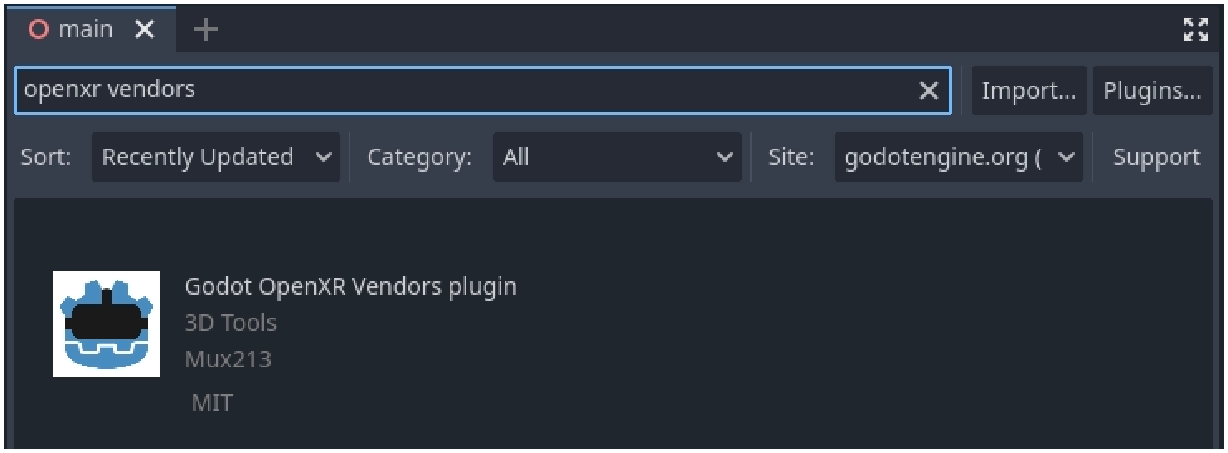
You can read more about gradle builds here: Gradle builds for Android.

Installing the vendors plugin

Warning

The Android plugin structure has been restructured in Godot 4.2, and the loader plugin was renamed to vendors plugin as it now includes more than just loaders. If you've previously installed the loader plugin you need to delete it from the *android/plugins* folder.

The vendors plugin can be downloaded from the asset library, search for "OpenXR vendors" and install the plugin:


main
openxr vendors
Import...
Plugins...
Sort: Recently Updated
Category: All
Site: godotengine.org (
Support
Godot OpenXR Vendors plugin
3D Tools
Mux213
MIT

You will find the installed files inside the **addons** folder.

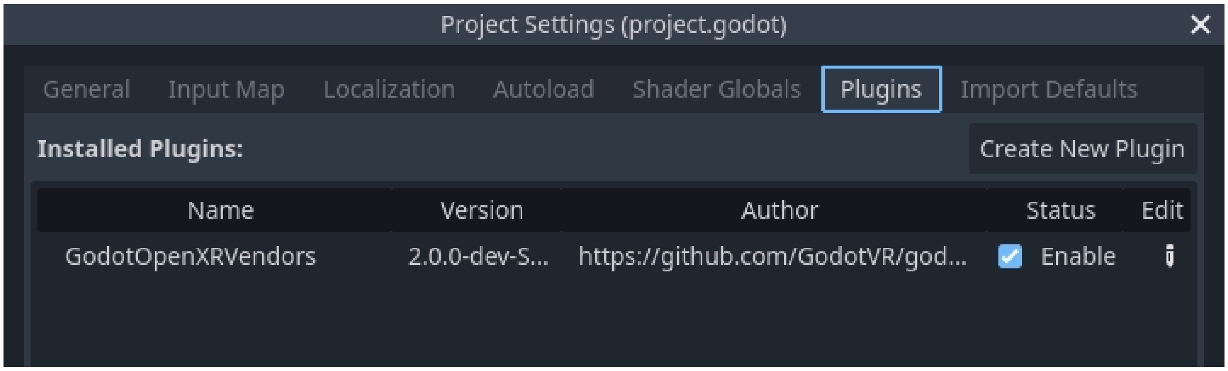
Alternatively you can manually install the vendors plugin by downloading the v2.x version of the plugin from the release page here [https://github.com/GodotVR/godot_openxr_vendors/releases].

You will need to copy the *assets/addons/godotopenxrvendors* folder from the zip file into your projects *addons* folder.

You can find the main repository of the vendors plugin here [https://github.com/GodotVR/godot_openxr_vendors].

Enabling the vendors plugin

The vendors plugin needs to be enabled before the export settings become accessible. Open **Project** and select **Project Settings...**. Go to the **Plugins** tab. Enable the **GodotOpenXRVendors** plugin.


Project Settings (project.godot)

Installed Plugins:

Name	Version	Author	Status	Edit
GodotOpenXRVendors	2.0.0-dev-S...	https://github.com/GodotVR/god...	☑ Enable	

Note

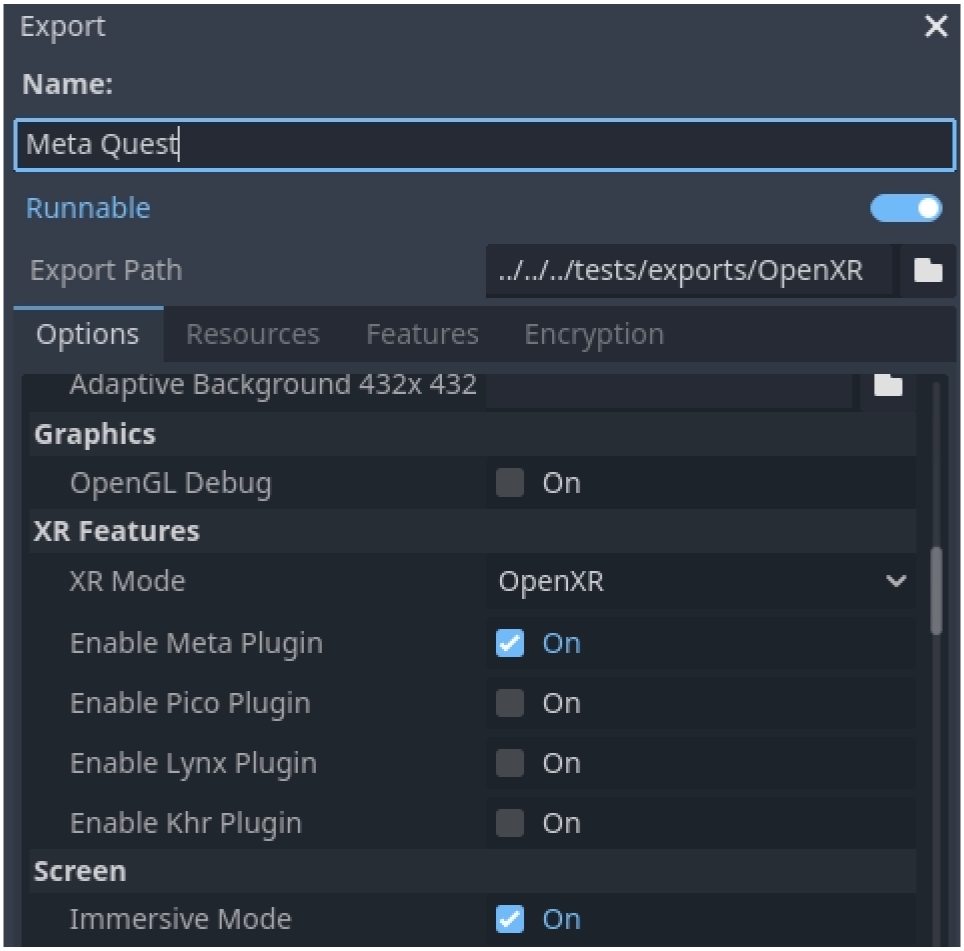
This is no longer required from vendors plugin 2.0.3 onwards as it now uses GDExtension. The plugin will not be shown in this list. You can verify it is installed correctly by checking if the export presets contain the entries described below.

Creating the export presets

You will need to setup a separate export preset for each device, as each device will need its own loader included.

Open **Project** and select **Export..**. Click on **Add..** and select **Android**. Next change the name of the export preset for the device you're setting this up for, say **Meta Quest**. And enable **Use Gradle Build**. If you want to use one-click deploy (described below), ensure that **Runnable** is enabled.

If the vendors plugins were installed correctly you should find entries for the different headsets, select the entry for meta:



Name: Meta Quest

Runnable ● On

Export Path ../../../tests/exports/OpenXR

Adaptive Background 432x 432

Graphics

OpenGL Debug ☐ On

XR Features

XR Mode OpenXR

Enable Meta Plugin ☑ On

Enable Pico Plugin ☐ On

Enable Lynx Plugin ☐ On

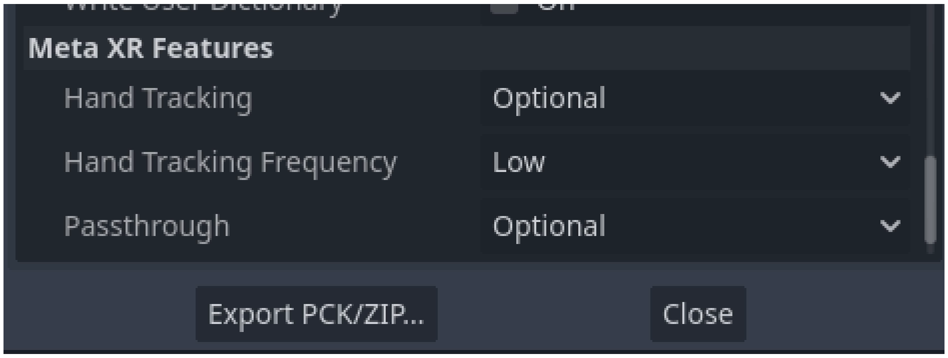
Enable Khr Plugin ☐ On

Screen

Immersive Mode ☑ On

Also change the **XR Mode** to **OpenXR**.

Scroll to the bottom of the list and you'll find additional XR feature sections, currently only **Meta XR Features** is available. The hand tracking and passthrough settings here currently only work for the Meta Quest and you will need to select the appropriate settings if you wish to use these features.



Meta XR Features

Hand Tracking Optional

Hand Tracking Frequency Low

Passthrough Optional

Now you can repeat the same process for the other devices.

Note

There are separate loaders for the Meta Quest, Pico and Lynx R1 headsets.

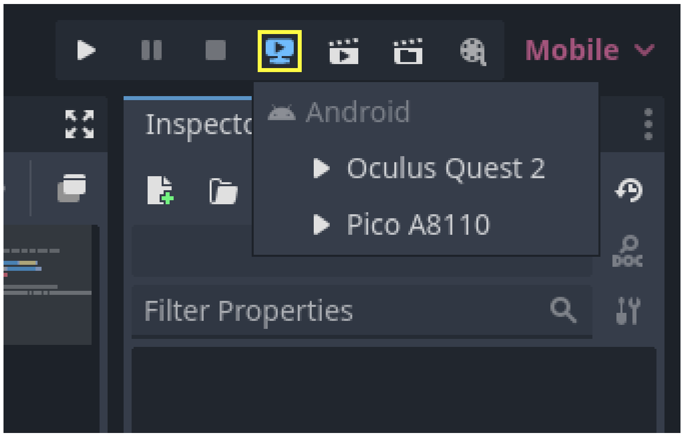
The fourth option is the official Khronos loader, in due time all headsets should work with this loader. At the moment this loader has been tested with the Magic Leap 2 and standalone HTC headsets.

Warning

While the Mobile Vulkan renderer has many optimizations targeted at mobile devices, we're still working out the kinks. It is highly advisable to use the compatibility renderer (OpenGL) for the time being when targeting Android based XR devices.

Running on your device from the Godot editor

If you've setup your export settings as described above, and your headset is connected to your computer and correctly recognized, you can launch it directly from the Godot editor using One-click deploy:


Mobile
Inspector
Android
Oculus Quest 2
Pico A8110
Filter Properties

For some devices on some platforms, you may need to perform some extra steps in order for your device to be recognized correctly, so be sure to check the developer documentation from your headset vendor.

For example, with the Meta Quest 2, you need to enable developer mode on the headset, and if you're on Windows, you'll need to install special ADB drivers. See the official Meta Quest developer documentation [https://developer.oculus.com/documentation/native/android/mobile-device-setup/] for more details.

If you're having any issues with one-click deploy, check the Troubleshooting section.

A better XR start script

In Setting up XR we introduced a startup script that initialises our setup which we used as our script on our main node. This script performs the minimum steps required for any given interface.

When using OpenXR there are a number of improvements we should do here. For this we've created a more elaborate starting script. You will find these used in our demo projects.

Alternatively, if you are using XR Tools (see Introducing XR tools) it contains a version of this script updated with some features related to XR tools.

Below we will detail out the script used in our demos and explain the parts that are added.

Signals for our script

We are introducing 3 signals to our script so that our game can add further logic:

- `focus_lost` is emitted when the player takes off their headset or when the player enters the menu system of the headset.
- `focus_gained` is emitted when the player puts their headset back on or exists the menu system and returns to the game.
- `pose_recentered` is emitted when the headset requests the players position to be reset.

Our game should react accordingly to these signals.

GDScript

```
extends Node3D

signal focus_lost
signal focus_gained
signal pose_recentered

...
```

C#

```
using Godot;

public partial class MyNode3D : Node3D
{
    [Signal]
    public delegate void FocusLostEventHandler();

    [Signal]
    public delegate void FocusGainedEventHandler();

    [Signal]
    public delegate void PoseRecenteredEventHandler();

...
```

Variables for our script

We introduce a few new variables to our script as well:

- `maximum_refresh_rate` will control the headsets refresh rate if this is supported by the headset.
- `xr_interface` holds a reference to our XR interface, this already existed but we now type it to get full access to our XRInterface API.
- `xr_is_focussed` will be set to true whenever our game has focus.

GDScript

```
...

@export var maximum_refresh_rate : int = 90

var xr_interface : OpenXRInterface
var xr_is_focussed = false

...
```

C#

```
...

    [Export]
    public int MaximumRefreshRate { get; set; } = 90;

    private OpenXRInterface _xrInterface;

    private bool _xrIsFocused;

...
```

Our updated ready function

We add a few things to the ready function.

If we're using the mobile or forward+ renderer we set the viewports `vrs_mode` to `VRS_XR`. On platforms that support this, this will enable foveated rendering.

If we're using the compatibility renderer, we check if the OpenXR foveated rendering settings are configured and if not, we output a warning. See OpenXR Settings for further details.

We hook up a number of signals that will be emitted by the XRInterface. We'll provide more detail about these signals as we implement them.

We also quit our application if we couldn't successfully initialise OpenXR. Now this can be a choice. If you are making a mixed mode game you setup the VR mode of your game on success, and setup the non-VR mode of your game on failure. However, when running a VR only application on a standalone headset, it is nicer to exit on failure than to hang the system.

GDScript

```
...

# Called when the node enters the scene tree for the first
time.
func _ready():
    xr_interface = XRServer.find_interface("OpenXR")
    if xr_interface and xr_interface.is_initialized():
        print("OpenXR instantiated successfully.")
        var vp : Viewport = get_viewport()

        # Enable XR on our viewport
        vp.use_xr = true

        # Make sure v-sync is off, v-sync is handled by OpenXR

DisplayServer.window_set_vsync_mode(DisplayServer.VSYNC_DISABLE
D)

        # Enable VRS
        if RenderingServer.get_rendering_device():
            vp.vrs_mode = Viewport.VRS_XR
        elif
int(ProjectSettings.get_setting("xr/openxr/foveation_level"))
== 0:
            push_warning("OpenXR: Recommend setting Foveation
level to High in Project Settings")

        # Connect the OpenXR events

xr_interface.session_begun.connect(_on_openxr_session_begun)

xr_interface.session_visible.connect(_on_openxr_visible_state)
xr_interface.session_focussed.connect(_on_openxr_focused_state)

xr_interface.session_stopping.connect(_on_openxr_stopping)

xr_interface.pose_recentered.connect(_on_openxr_pose_recentered
)
    else:
        # We couldn't start OpenXR.
        print("OpenXR not instantiated!")
        get_tree().quit()

...
```

C#

```
...

    /// <summary>
    /// Called when the node enters the scene tree for the
first time.
    /// </summary>
    public override void _Ready()
    {
        _xrInterface =
(OpenXRInterface)XRServer.FindInterface("OpenXR");
        if (_xrInterface != null &&
_xrInterface.IsInitialized())
        {
            GD.Print("OpenXR instantiated successfully.");
            var vp = GetViewport();

            // Enable XR on our viewport
            vp.UseXR = true;

            // Make sure v-sync is off, v-sync is handled by
OpenXR

DisplayServer.WindowSetVsyncMode(DisplayServer.VSyncMode.Disabl
ed);

            // Enable VRS
            if (RenderingServer.GetRenderingDevice() != null)
                vp.VrsMode = Viewport.VrsModeEnum.XR;
            else if
((int)ProjectSettings.GetSetting("xr/openxr/foveation_level")
== 0)
                GD.PushWarning("OpenXR: Recommend setting
Foveation level to High in Project Settings");

            // Connect the OpenXR events
            _xrInterface.SessionBegun += OnOpenXRSessionBegun;
            _xrInterface.SessionVisible +=
OnOpenXRVisibleState;
            _xrInterface.SessionFocussed +=
OnOpenXRFocusedState;
            _xrInterface.SessionStopping += OnOpenXRStopping;
            _xrInterface.PoseRecentered +=
OnOpenXRPoseRecentered;
        }
        else
        {
            // We couldn't start OpenXR.
            GD.Print("OpenXR not instantiated!");
            GetTree().Quit();
        }
    }

...
```

On session begun

This signal is emitted by OpenXR when our session is setup. This means the headset has run through setting everything up and is ready to begin receiving content from us. Only at this time various information is properly available.

The main thing we do here is to check our headsets refresh rate. We also check the available refresh rates reported by the XR runtime to determine if we want to set our headset to a higher refresh rate.

Finally we match our physics update rate to our headset update rate. Godot runs at a physics update rate of 60 updates per second by default while headsets run at a minimum of 72, and for modern headsets often up to 144 frames per second. Not matching the physics update rate will cause stuttering as frames are rendered without objects moving.

GDScript

```
...

# Handle OpenXR session ready
func _on_openxr_session_begun() -> void:
    # Get the reported refresh rate
    var current_refresh_rate =
xr_interface.get_display_refresh_rate()
    if current_refresh_rate > 0:
        print("OpenXR: Refresh rate reported as ",
str(current_refresh_rate))
    else:
        print("OpenXR: No refresh rate given by XR runtime")

    # See if we have a better refresh rate available
    var new_rate = current_refresh_rate
    var available_rates : Array =
xr_interface.get_available_display_refresh_rates()
    if available_rates.size() == 0:
        print("OpenXR: Target does not support refresh rate
extension")
    elif available_rates.size() == 1:
        # Only one available, so use it
        new_rate = available_rates[0]
    else:
        for rate in available_rates:
            if rate > new_rate and rate <=
maximum_refresh_rate:
                new_rate = rate

    # Did we find a better rate?
    if current_refresh_rate != new_rate:
        print("OpenXR: Setting refresh rate to ",
str(new_rate))
        xr_interface.set_display_refresh_rate(new_rate)
        current_refresh_rate = new_rate

    # Now match our physics rate
    Engine.physics_ticks_per_second = current_refresh_rate

...
```

C#

```
...

    /// <summary>
    /// Handle OpenXR session ready
    /// </summary>
    private void OnOpenXRSessionBegun()
    {
        // Get the reported refresh rate
        var currentRefreshRate =
_xrInterface.DisplayRefreshRate;
        GD.Print(currentRefreshRate > 0.0F
            ? $"OpenXR: Refresh rate reported as
{currentRefreshRate}"
            : "OpenXR: No refresh rate given by XR runtime");

        // See if we have a better refresh rate available
        var newRate = currentRefreshRate;
        var availableRates =
_xrInterface.GetAvailableDisplayRefreshRates();
        if (availableRates.Count == 0)
        {
            GD.Print("OpenXR: Target does not support refresh
rate extension");
        }
        else if (availableRates.Count == 1)
        {
            // Only one available, so use it
            newRate = (float)availableRates[0];
        }
        else
        {
            GD.Print("OpenXR: Available refresh rates: ",
availableRates);
            foreach (float rate in availableRates)
                if (rate > newRate && rate <=
MaximumRefreshRate)
                    newRate = rate;
        }

        // Did we find a better rate?
        if (currentRefreshRate != newRate)
        {
            GD.Print($"OpenXR: Setting refresh rate to
{newRate}");
            _xrInterface.DisplayRefreshRate = newRate;
            currentRefreshRate = newRate;
        }

        // Now match our physics rate
        Engine.PhysicsTicksPerSecond = (int)currentRefreshRate;
    }

...
```

On visible state

This signal is emitted by OpenXR when our game becomes visible but is not focussed. This is a bit of a weird description in OpenXR but it basically means that our game has just started and we're about to switch to the focussed state next, that the user has opened a system menu or the users has just took their headset off.

On receiving this signal we'll update our focussed state, we'll change the process mode of our node to disabled which will pause processing on this node and it's children, and emit our `focus_lost` signal.

If you've added this script to your root node, this means your game will automatically pause when required. If you haven't, you can connect a method to the signal that performs additional changes.

Note

While your game is in visible state because the user has opened a system menu, Godot will keep rendering frames and head tracking will remain active so your game will remain visible in the background. However controller and hand tracking will be disabled until the user exits the system menu.

GDScript

```
...

# Handle OpenXR visible state
func _on_openxr_visible_state() -> void:
    # We always pass this state at startup,
    # but the second time we get this it means our player took
off their headset
    if xr_is_focussed:
        print("OpenXR lost focus")

        xr_is_focussed = false

        # pause our game
        get_tree().paused = true

        emit_signal("focus_lost")

...
```

C#

```
...

    /// <summary>
    /// Handle OpenXR visible state
    /// </summary>
    private void OnOpenXRVisibleState()
    {
        // We always pass this state at startup,
        // but the second time we get this it means our player
took off their headset
        if (_xrIsFocused)
        {
            GD.Print("OpenXR lost focus");

            _xrIsFocused = false;

            // Pause our game
            GetTree().Paused = true;

            EmitSignal(SignalName.FocusLost);
        }
    }

...
```

On focussed state

This signal is emitted by OpenXR when our game gets focus. This is done at the completion of our startup, but it can also be emitted when the user exits a system menu, or put their headset back on.

Note also that when your game starts while the user is not wearing their headset, the game stays in 'visible' state until the user puts their headset on.

Warning

It is thus important to keep your game paused while in visible mode. If you don't the game will keep on running while your user isn't interacting with your game. Also when the game returns to focussed mode, suddenly all controller and hand tracking is re-enabled and could have game breaking consequences if you do not react to this accordingly. Be sure to test this behaviour in your game!

While handling our signal we will update the focusses state, unpause our node and emit our `focus_gained` signal.

GDScript

```
...

# Handle OpenXR focused state
func _on_openxr_focused_state() -> void:
    print("OpenXR gained focus")
    xr_is_focussed = true

    # unpause our game
    get_tree().paused = false

    emit_signal("focus_gained")

...
```

C#

```
...

    /// <summary>
    /// Handle OpenXR focused state
    /// </summary>
    private void OnOpenXRFocusedState()
    {
        GD.Print("OpenXR gained focus");
        _xrIsFocused = true;

        // Un-pause our game
        GetTree().Paused = false;

        EmitSignal(SignalName.FocusGained);
    }
```

```
...
```

On stopping state

This signal is emitted by OpenXR when we enter our stop state. There are some differences between platforms when this happens. On some platforms this is only emitted when the game is being closed. But on other platforms this will also be emitted every time the player takes off their headset.

For now this method is only a place holder.

GDScript

```
...

# Handle OpenXR stopping state
func _on_openxr_stopping() -> void:
    # Our session is being stopped.
    print("OpenXR is stopping")

...
```

C#

```
...

    /// <summary>
    /// Handle OpenXR stopping state
    /// </summary>
    private void OnOpenXRStopping()
    {
        // Our session is being stopped.
        GD.Print("OpenXR is stopping");
    }

...
```

On pose recentered

This signal is emitted by OpenXR when the user requests their view to be recentered. Basically this communicates to your game that the user is now facing forward and you should re-orient the player so they are facing forward in the virtual world.

As doing so is dependent on your game, your game needs to react accordingly.

All we do here is emit the `pose_recentered` signal. You can connect to this signal and implement the actual recenter code. Often it is enough to call center_on_hmd().

GDScript

```
...

# Handle OpenXR pose recentered signal
func _on_openxr_pose_recentered() -> void:
    # User recentered view, we have to react to this by
recentering the view.
    # This is game implementation dependent.
    emit_signal("pose_recentered")
```

C#

```
...

    /// <summary>
    /// Handle OpenXR pose recentered signal
    /// </summary>
    private void OnOpenXRPoseRecentered()
    {
        // User recentered view, we have to react to this by
recentering the view.
        // This is game implementation dependent.
        EmitSignal(SignalName.PoseRecentered);
        }
}
```

And that finished our script. It was written so that it can be re-used over multiple projects. Just add it as the script on your main node (and extend it if needed) or add it on a child node specific for this script.

AR / Passthrough

Augmented Reality is supported through various methods depending on the capabilities of the hardware.

Headsets such as the Magic Leap and glasses such as TiltFive show the rendered result on see-through displays [https://en.wikipedia.org/wiki/See-through_display] allowing the user to see the real world.

Headsets such as the Quest, HTC Elite, and Lynx R1 implement this through a technique called video passthrough, where cameras record the real world and these images are used as the background on top of which our rendered result is used.

Note

Passthrough is implemented very differently across platforms.

In Godot 4.3 we have implemented a unified approach that is explained on this help page so you don't need to worry about these differences, the XRInterface implementation is now responsible for applying the correct platform dependent method [1].

For headsets such as the Meta Quest and HTC Elite you will need to use the OpenXR vendors plugin v3.0.0 [https://github.com/GodotVR/godot_openxr_vendors/releases] or later to enable video passthrough.

For backwards compatibility the old API for passthrough is still available but it is recommended to follow the new instructions below.

Environment blend modes

The way we configure VR or AR functionality is through setting the environment blend mode. This mode determines how the (real world) environment is blended with the virtual world.

Blend modes

Blend mode	Description
XR_ENV_BLEND_MODE_OPAQUE	The rendered image is opaque, we do not see the real world. We're in VR mode. This will turn off passthrough if video-passthrough is used.
XR_ENV_BLEND_MODE_ADDITIVE	The rendered image is added to the real world and will look semi transparent. This mode is generally used with see-through devices that are unable to obscure the real world. This will turn on passthrough if video-passthrough is used.
XR_ENV_BLEND_MODE_ALPHA_BLEND	The rendered image is alpha blended with the real world. On see-through devices that support this, the alpha will

Blend mode	Description
	control the translucency of the optics. On video-passthrough devices alpha blending is applied with the video image. passthrough will also be enabled if applicable.

You can set the environment blend mode for your application through the `environment_blend_mode` property of the XRInterface instance.

You can query the supported blend modes on the hardware using the `get_supported_environment_blend_modes` property on the same instance.

Configuring your background

When setting the blend mode to `XR_ENV_BLEND_MODE_ALPHA_BLEND` you must set the `transparent_bg` property on Viewport to true. When using the `XR_ENV_BLEND_MODE_ADDITIVE` blend mode you should set your background color to black.

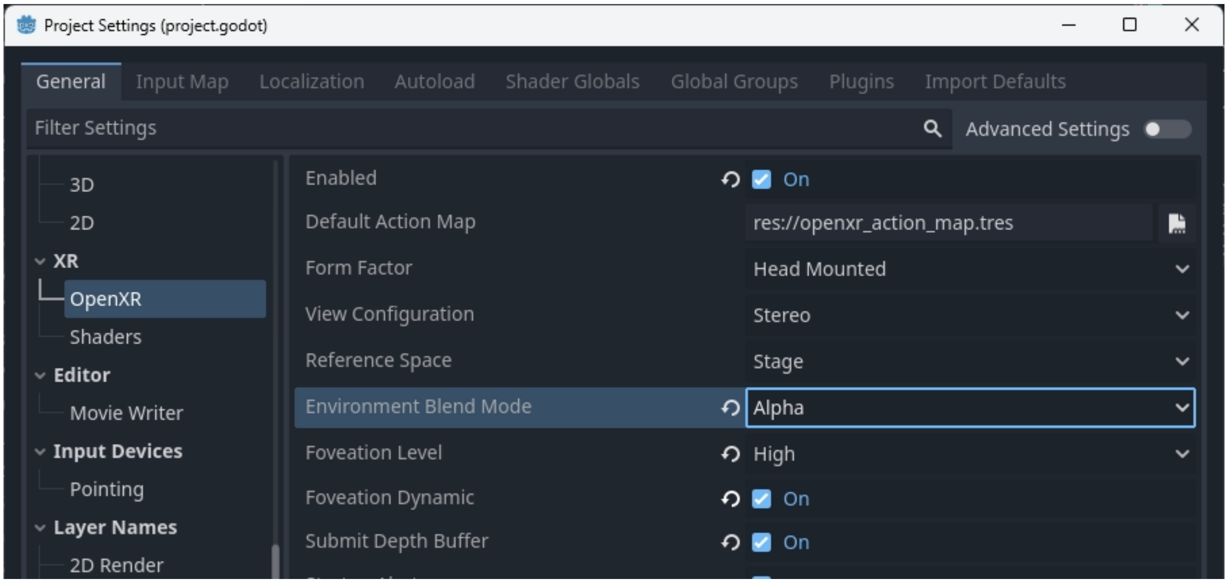
Either solution will result in the background rendering not contributing to lighting. It is thus also recommended you adjust your environment settings accordingly and ensure there is adequate ambient light set to illuminate your scene.

Note

Some AR SDKs do provide ambient lighting information or even provide a full radiance map to allow for real world reflections in your virtual objects. The core Godot XR functionality doesn't currently have support for this, however this functionality can be exposed through plugins.

OpenXR specific

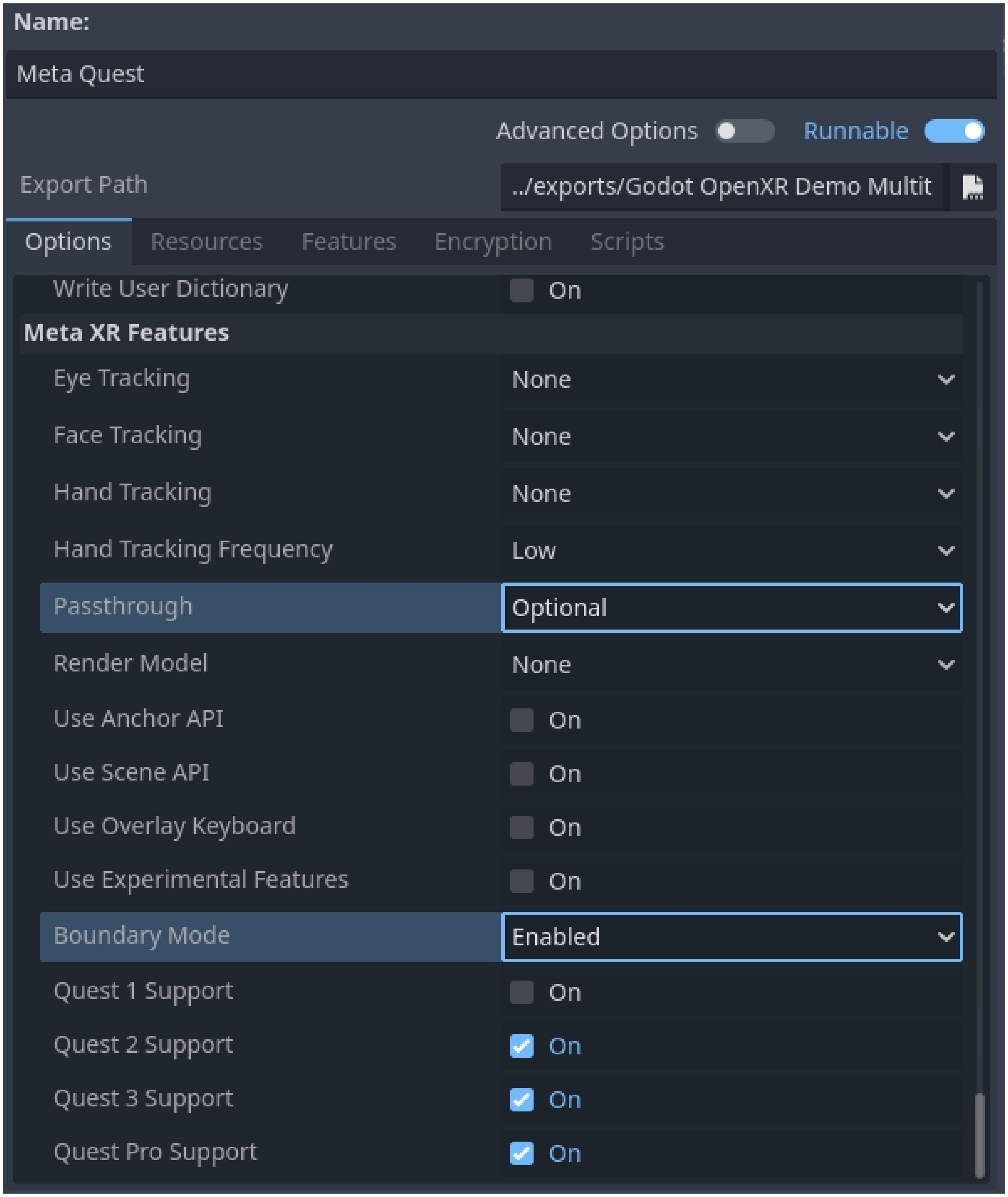
In OpenXR you can configure the default blend mode you want to use. Godot will select this blend mode at startup if available. If not available Godot will default to the first supported blend mode provided by the XR runtime.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Import Defaults
Filter Settings
Advanced Settings
3D
2D
XR
OpenXR
Shaders
Editor
Movie Writer
Input Devices
Pointing
Layer Names
2D Render

Enabled	☑ On
Default Action Map	res://openxr_action_map.tres
Form Factor	Head Mounted
View Configuration	Stereo
Reference Space	Stage
Environment Blend Mode	Alpha
Foveation Level	High
Foveation Dynamic	☑ On
Submit Depth Buffer	☑ On

For passthrough devices OpenXR requires additional settings to be configured. These settings are platform dependent and provided through the OpenXR vendors plugin.

For example, these are the settings required on Meta Quest:



Name:

Meta Quest

Advanced Options
Runnable
Options
Resources
Features
Encryption
Scripts

Export Path: ../exports/Godot OpenXR Demo Multit

Write User Dictionary ☐ On

Meta XR Features

Eye Tracking: None

Face Tracking: None

Hand Tracking: None

Hand Tracking Frequency: Low

Passthrough: Optional

Render Model: None

Use Anchor API ☐ On

Use Scene API ☐ On

Use Overlay Keyboard ☐ On

Use Experimental Features ☐ On

Boundary Mode: Enabled

Quest 1 Support ☐ On

Quest 2 Support ☑ On

Quest 3 Support ☑ On

Quest Pro Support ☑ On

The `Passthrough` setting defines whether passthrough is supported or even required.

The `Boundary Mode` allows you to define whether the guardian is needed, disabling this fully requires passthrough to be enabled at all times.

Putting it together

Putting the above together we can use the following code as a base:

```
@onready var viewport : Viewport = get_viewport()
@onready var environment : Environment =
$WorldEnvironment.environment

func switch_to_ar() -> bool:
    var xr_interface: XRInterface = XRServer.primary_interface
    if xr_interface:
        var modes =
xr_interface.get_supported_environment_blend_modes()
        if XRInterface.XR_ENV_BLEND_MODE_ALPHA_BLEND in modes:
            xr_interface.environment_blend_mode =
XRInterface.XR_ENV_BLEND_MODE_ALPHA_BLEND
            viewport.transparent_bg = true
        elif XRInterface.XR_ENV_BLEND_MODE_ADDITIVE in modes:
            xr_interface.environment_blend_mode =
XRInterface.XR_ENV_BLEND_MODE_ADDITIVE
            viewport.transparent_bg = false
    else:
        return false

    environment.background_mode = Environment.BG_COLOR
    environment.background_color = Color(0.0, 0.0, 0.0, 0.0)
    environment.ambient_light_source =
Environment.AMBIENT_SOURCE_COLOR
    return true

func switch_to_vr() -> bool:
    var xr_interface: XRInterface = XRServer.primary_interface
    if xr_interface:
        var modes =
xr_interface.get_supported_environment_blend_modes()
        if XRInterface.XR_ENV_BLEND_MODE_OPAQUE in modes:
            xr_interface.environment_blend_mode =
XRInterface.XR_ENV_BLEND_MODE_OPAQUE
        else:
            return false

    viewport.transparent_bg = false
    environment.background_mode = Environment.BG_SKY
    environment.ambient_light_source =
Environment.AMBIENT_SOURCE_BG
    return true
```

Shadow to opacity

Shadow to opacity is a render mode for Godot spatial shaders that was introduced in Godot 3 specifically for AR. It is a special render mode where the more a surface is in shadow, the more opaque the surface becomes. When a surface is fully lit, the surface becomes fully transparent and thus shows the real world.

However the surface is rendered during the opaque state effectively. This has two consequences:

- As both the depth buffer and color buffer are written to, we occlude any geometry behind our surface even when fully transparent.
- As we are making the surface opaque if in shadow, we can have virtual objects cast shadows on real world objects [2].

Image showing shadow to opacity being used to show the users desk.

This enabled the following use cases:

- You can render a box mesh around a real world table, this ensures the table remains visible even if a virtual object is placed underneath it. The virtual object will be correctly occluded. Placing a virtual object on top of the real world table, will result in a shadow being cast on the table.
- You can use a shader with this render mode when render a hand mesh using the hand tracking functionality, and ensure your hands properly occlude virtual objects.

The following shader code is a good base for this functionality:

```
shader_type spatial;
render_mode blend_mix, depth_draw_opaque, cull_back,
shadow_to_opacity;

void fragment() {
    ALBEDO = vec3(0.0, 0.0, 0.0);
}
```

[1] Restrictions may apply depending on XR interface implementation.

[2] This feature is still being perfected.

Where to go from here

Now that we have the basics covered there are several options to look at for your XR game dev journey:

- You can take a look at the Advanced topics section.
- You can look at a number of XR demos here [https://github.com/godotengine/godot-demo-projects/tree/master/xr].
- You can find 3rd party tutorials on our Tutorials and resources page.

XR Toolkits

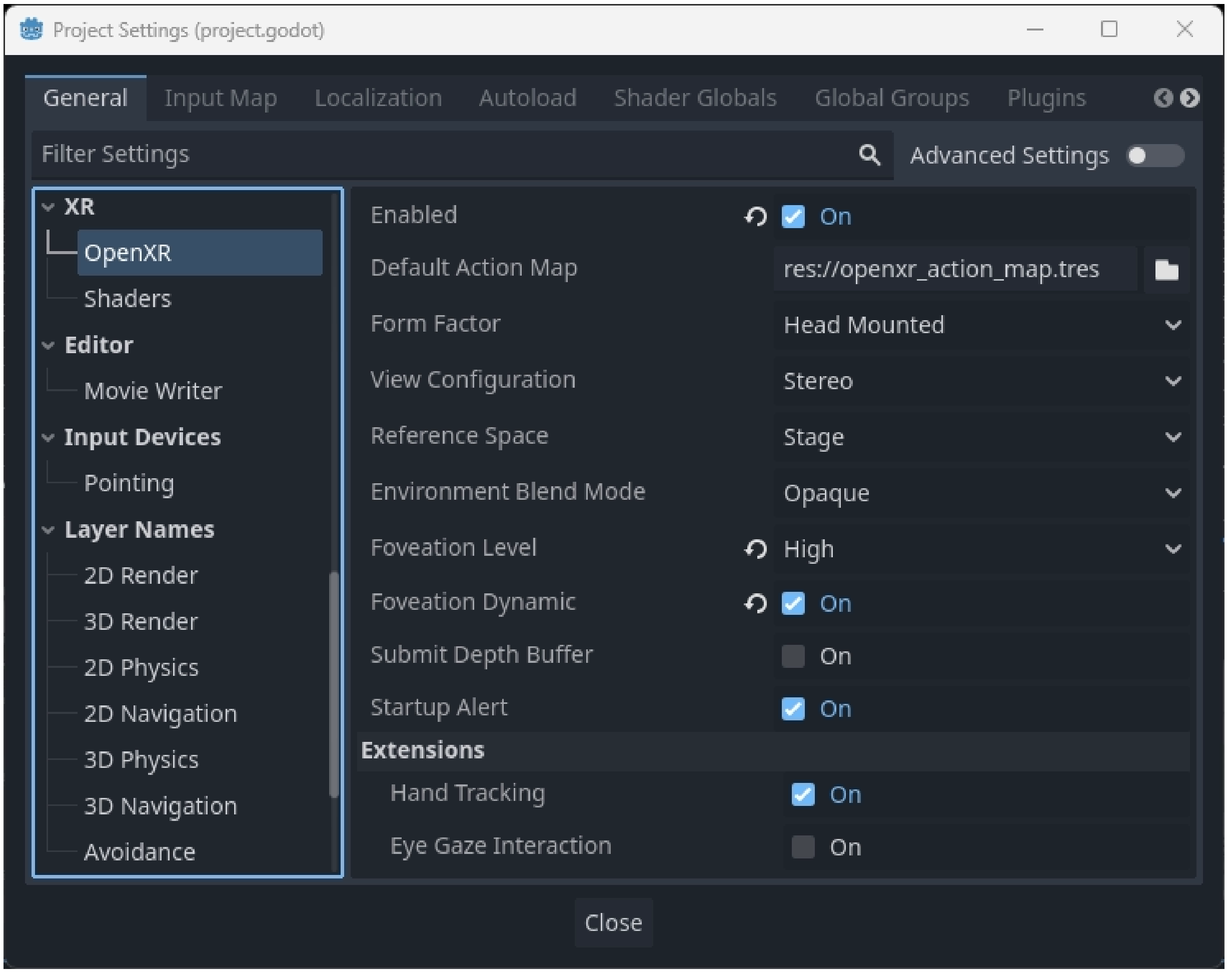
There are various XR toolkits available that implement more complex XR logic ready for you to use. We have a small introduction to Godot XR Tools that you can look at, a toolkit developed by core contributors of Godot.

There are more toolkits available for Godot:

- Godot XR handtracking toolkit [https://github.com/RevolNoom/godot_xr_handtracking] (GDScript)
- Godot XR Kit [https://github.com/patrykkalinowski/godot-xr-kit] (GDScript)
- Godot XR Tools [https://github.com/godotvr/godot-xr-tools] (GDScript)
- NXR [https://github.com/stumpynub/NXR] (C#)

OpenXR Settings

OpenXR has its own set of settings that are applied when OpenXR starts. While it is possible for OpenXR extensions implemented through Godot plugins to add additional settings, we will only discuss the settings in the core of Godot here.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Filter Settings
Advanced Settings
XR
OpenXR
Shaders
Editor
Movie Writer
Input Devices
Pointing
Layer Names
2D Render
3D Render
2D Physics
2D Navigation
3D Physics
3D Navigation
Avoidance
Close

Enabled ☑ On

Default Action Map res://openxr_action_map.tres

Form Factor Head Mounted

View Configuration Stereo

Reference Space Stage

Environment Blend Mode Opaque

Foveation Level High

Foveation Dynamic ☑ On

Submit Depth Buffer ☐ On

Startup Alert ☑ On

Extensions

Hand Tracking ☑ On

Eye Gaze Interaction ☐ On

Enabled

This setting enables the OpenXR module when Godot starts. This is required when the Vulkan backend is used. For other backends you can enable OpenXR at any time by calling `initialize` on the OpenXRInterface.

This also needs to be enabled to get access to the action map editor.

You can use the `--xr-mode on` command line switch to force this to on.

Default Action Map

This specifies the path of the action map file that OpenXR will load and communicate to the XR Runtime.

Form Factor

This specifies whether your game is designed for:

- `Head Mounted` devices such as a Meta Quest, Valve Index, or Magic Leap,
- `Handheld` devices such as phones.

If the device on which you run your game does not match the selection here, OpenXR will fail to initialise.

View Configuration

This specifies the view configuration your game is designed for:

- `Mono`, your game provides a single image output. E.g. phone based AR;
- `Stereo`, your game provides stereo image output. E.g. head mounted devices.

If the device on which you run your game does not match the selection here, OpenXR will fail to initialise.

Note

OpenXR has additional view configurations for very specific devices that Godot doesn't support yet. For instance, Varjo headsets have a quad view configuration that outputs two sets of stereo images. These may be supported in the near future.

Reference Space

Within XR all elements like the player's head and hands are tracked within a tracking volume. At the base of this tracking volume is our origin point, which maps our virtual space to the real space. There are however different scenarios that place this point in different locations, depending on the XR system used. In OpenXR these scenarios are well defined and selected by setting a reference space.

Local

The local reference space places our origin point at the player's head by default. Some XR runtimes will do this each time your game starts, others will make the position persist over sessions.

This reference space however does not prevent the user from walking away so you will need to detect if the user does so if you wish to prevent the user from leaving the vehicle they are controlling, which could potentially be game breaking.

This reference space is the best option for games like flight simulators or racing simulators where we want to place the XROrigin3D node where the player's head should be.

When the user enacts the recenter option on their headset, the method of which is different per XR runtime, the XR runtime will move the XRCamera3D to the XROrigin3D node. The OpenXRInterface will also emit the `pose_recentered` signal so your game can react accordingly.

Note

Any other XR tracked elements such as controllers or anchors will also be adjusted accordingly.

Warning

You should **not** call `center_on_hmd` when using this reference space.

Stage

The stage reference space is our default reference space and places our origin point at the center of our play space. For XR runtimes that allow you to draw out a guardian boundary this location and its orientation is often set by the user. Other XR runtimes may decide on the placement of this point by other means. It is however a stationary point in the real world.

This reference space is the best option for room scale games where the user is expected to walk around a larger space, or for games where there is a need to switch between game modes. See Room Scale for more information.

When the user enacts the recenter option on their headset, the method of which is different per XR runtime, the XR runtime will not change the origin point. The OpenXRInterface will emit the `pose_recentered` signal and it is up to the game to react appropriately. Not doing so will prevent your game from being accepted on various stores.

In Godot you can do this by calling the `center_on_hmd` function on the XRServer:

- Calling `XRServer.center_on_hmd(XRServer.RESET_BUT_KEEP_TILT, true)` will move the XRCamera3D node to the XROrigin3D node similar to the `Local` reference space.
- Calling `XRServer.center_on_hmd(XRServer.RESET_BUT_KEEP_TILT, true)` will move the XRCamera3D node above the XROrigin3D node keeping the player's height, similar to the `Local Floor` reference space.

Note

Any other XR tracked elements such as controllers or anchors will also be adjusted accordingly.

Local Floor

The local floor reference space is similar to the local reference space as it positions the origin point where the player is. In this mode however the height of the player is kept. Same as with the local reference space, some XR runtimes will persist this location over sessions.

It is thus not guaranteed the player will be standing on the origin point, the only guarantee is that they were standing there when the user last recentered. The player is thus also free to walk away.

This reference space is the best option of games where the user is expected to stand in the same location or for AR type games where the user's interface elements are bound to the origin node and are quickly placed at the player's location on recenter.

When the user enacts the recenter option on their headset, the method of which is different per XR runtime, the XR runtime will move the XRCamera3D above the XROrigin3D node but keeping the player's height. The OpenXRInterface will also emit the `pose_recentered` signal so your game can react accordingly.

Warning

Be careful using this mode in combination with virtual movement of the player. The user recentering in this scenario can be unpredictable unless you counter the move when handling the recenter signal. This can even be game breaking as the effect in this scenario would be the player teleporting to whatever abstract location the origin point was placed at during virtual movement, including the ability for players teleporting into locations that should be off limits. It is better to use the Stage mode in this scenario and limit resetting to orientation only when a `pose_recentered` signal is received.

Note

Any other XR tracked elements such as controllers or anchors will also be adjusted accordingly.

Warning

You should **not** call `center_on_hmd` when using this reference space.

Environment Blend Mode

The environment blend mode defines how our rendered output is blended into "the real world" provided this is supported by the headset.

- `Opaque` means our output obscures the real world, we are in VR mode.
- `Additive` means our output is added to the real world, this is an AR mode where optics do not allow us to fully obscure the real world (e.g. Hololens),
- `Alpha` means our output is blended with the real world using the alpha output (viewport should have transparent background enabled), this is an AR mode where optics can fully obscure the real world (Magic Leap, all pass through devices, etc.).

If a mode is selected that is not supported by the headset, the first available mode will be selected.

Note

Some OpenXR devices have separate systems for enabling/disabling passthrough. From Godot 4.3 onwards selecting the alpha blend mode will also perform these extra steps. This does require the latest vendor plugin to be installed.

Foveation Level

Sets the foveation level used when rendering provided this feature is supported by the hardware used. Foveation is a technique where the further away from the center of the viewport we render content, the lower resolution we render at. Most XR runtimes only support fixed foveation, but some will take eye tracking into account and use the focal point for this effect.

The higher the level, the better the performance gains, but also the more reduction in quality there is in the users peripheral vision.

Note

Compatibility renderer only, for Mobile and Forward+ renderer, set the `vrs_mode` property on Viewport to `VRS_XR`.

Warning

This feature is disabled if post effects are used such as glow, bloom, or DOF.

Foveation Dynamic

When enabled the foveation level will be adjusted automatically depending on current GPU load. It will be adjusted between low and the select foveation level in the previous setting. It is therefore best to combine this setting with foveation level set to high.

Note

Compatibility renderer only

Submit Depth Buffer

If enabled an OpenXR supplied depth buffer will be used while rendering which is submitted alongside the rendered image. The XR runtime can use this for improved reprojection.

Note

Enabling this feature will disable stencil support during rendering. Not many XR runtimes make use of this, it is advised to leave this setting off unless it provides noticeable benefits for your use case.

Startup Alert

If enabled, this will result in an alert message presented to the user if OpenXR fails to start. We don't always receive feedback from the XR system as to why starting fails. If we do, we log this to the console. Common failure reasons are:

- No OpenXR runtime is installed on the host system.
- Microsoft's WMR OpenXR runtime is currently active, this only supports DirectX and will fail if OpenGL or Vulkan is used.
- SteamVR is used but no headset is connected/turned on.

Disable this if you support a fallback mode in your game so it can be played in desktop mode when no VR headset is connected, or if you're handling the failure condition yourself by checking `OpenXRInterface.is_initialized()`.

Extensions

This subsection provides access to various optional OpenXR extensions.

Hand Tracking

This enables the hand tracking extension when supported by the device used. This is on by default for legacy reasons. The hand tracking extension provides access to data that allows you to visualise the user's hands with correct finger positions. Depending on platform capabilities the hand tracking data can be inferred from controller inputs, come from data gloves, come from optical hand tracking sensors or any other applicable source.

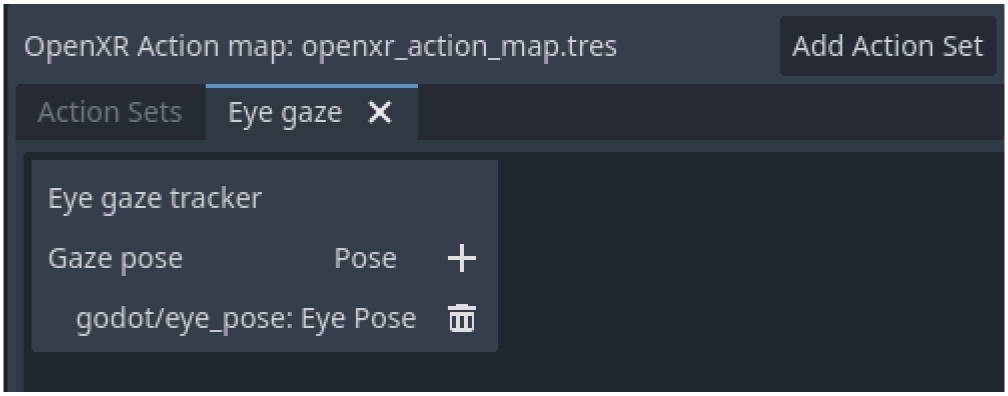
If your game only supports controllers this should be turned off.

See the chapter on hand tracking for additional details.

Eye Gaze Interaction

This enables the eye gaze interaction extension when supported by the device used. When enabled we will get feedback from eye tracking through a pose situated between the user's eyes orientated in the direction the user is looking. This will be a unified orientation.

In order to use this functionality you need to edit your action map and add a new pose action, say `eye_pose`. Now add a new interaction profile for the eye gaze interaction and map the `eye_pose`:


OpenXR Action map: openxr_action_map.tres
Add Action Set
Action Sets
Eye gaze
Eye gaze tracker
Gaze pose
Pose
godot/eye_pose: Eye Pose

Don't forget to save!

Next add a new XRController3D node to your origin node and set its `tracker` property to `/user/eyes_ext` and set its `pose` property to `eye_pose`.

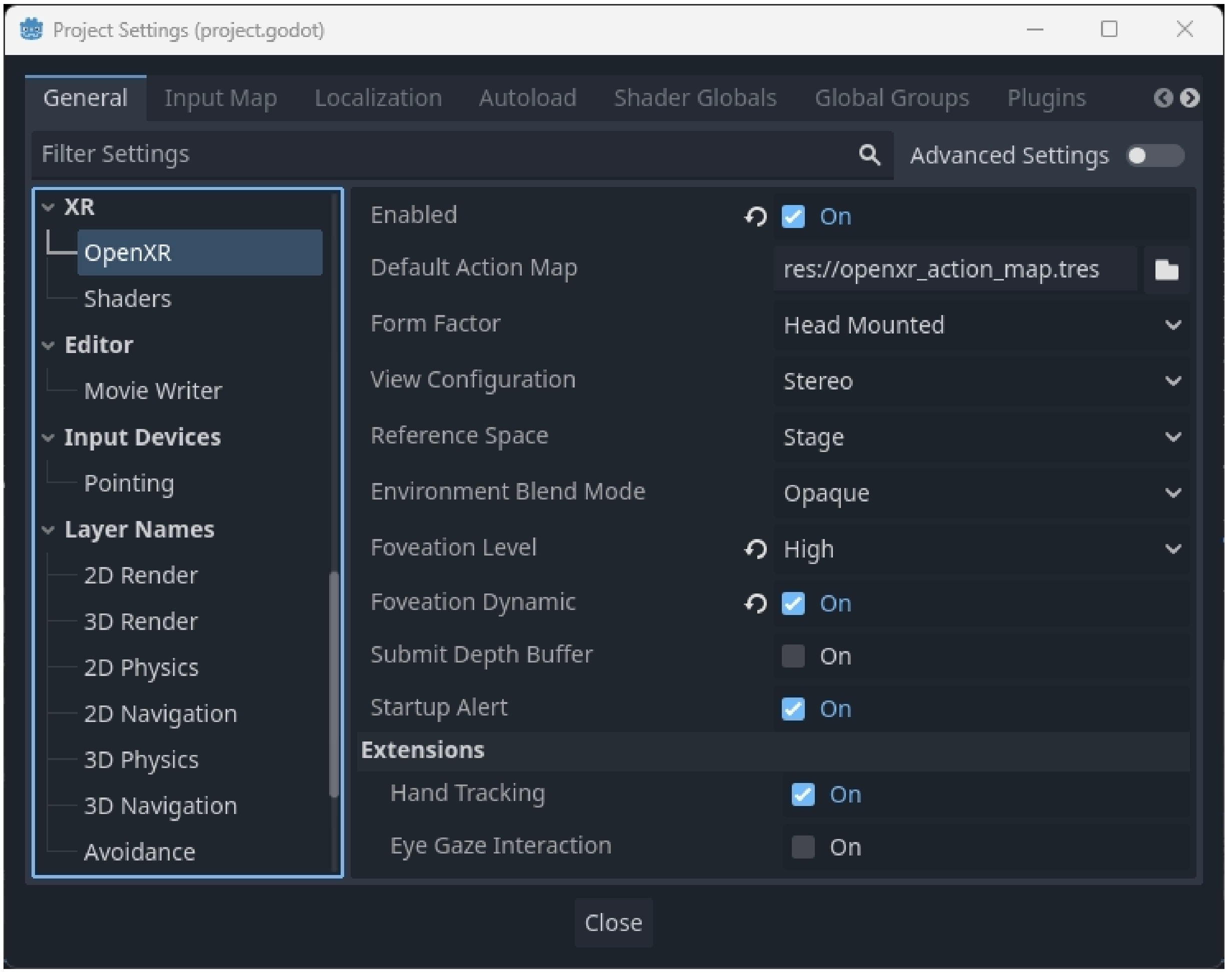
Now you can add things to this controller node such as a raycast, and control things with your eyes.

The XR action map

Godot has an action map feature as part of the XR system. At this point in time this system is part of the OpenXR module. There are plans to encompass WebXR into this in the near future hence we call it the XR action map system in this document. It implements the built-in action map system of OpenXR mostly exactly as it is offered.

The XR action map system exposes input, positional data and output for XR controllers to your game/application. It does this by exposing named actions that can be tailored to your game/application and binding these to the actual inputs and outputs on your XR devices.

As the XR action map is currently part of the OpenXR module, OpenXR needs to be enabled in your project settings to expose it:


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Filter Settings
Advanced Settings
XR
OpenXR
Shaders
Editor
Movie Writer
Input Devices
Pointing
Layer Names
2D Render
3D Render
2D Physics
2D Navigation
3D Physics
3D Navigation
Avoidance
Close

Enabled ☑ On

Default Action Map res://openxr_action_map.tres

Form Factor Head Mounted

View Configuration Stereo

Reference Space Stage

Environment Blend Mode Opaque

Foveation Level High

Foveation Dynamic ☑ On

Submit Depth Buffer ☐ On

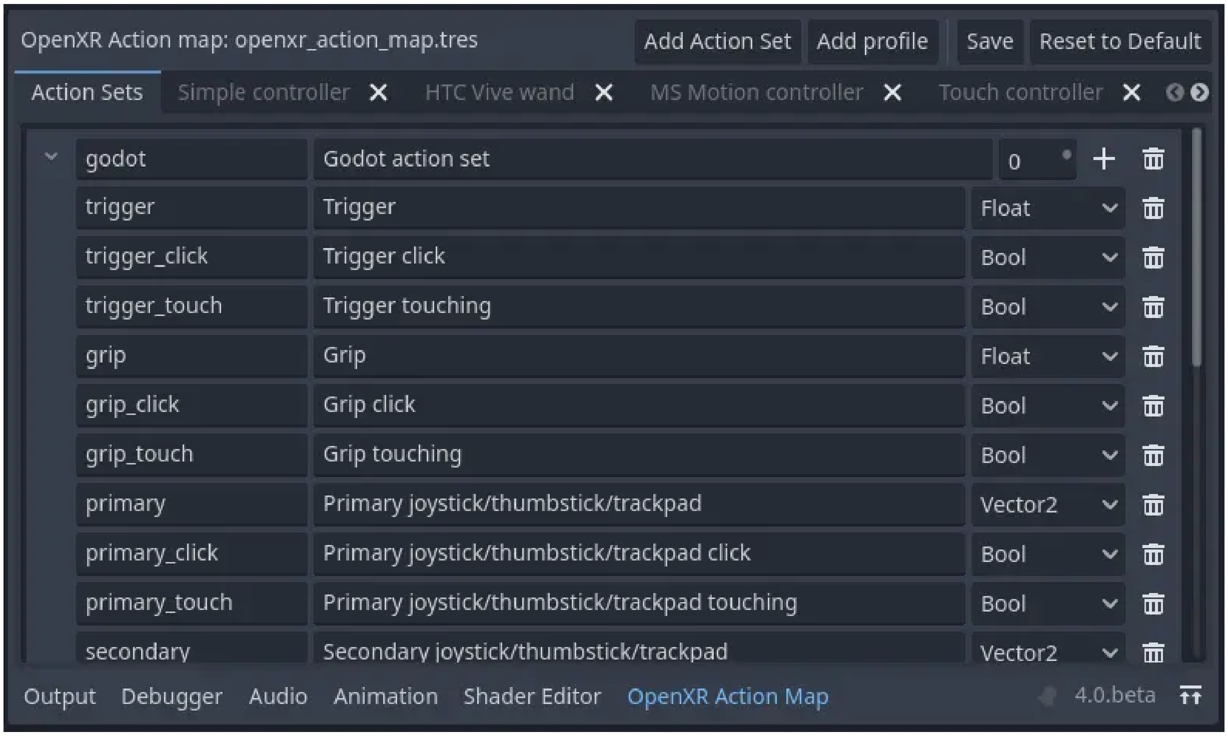
Startup Alert ☑ On

Extensions

Hand Tracking ☑ On

Eye Gaze Interaction ☐ On

You will then find the XR Action Map interface in the bottom of the screen:


OpenXR Action map: openxr_action_map.tres
Add Action Set
Add profile
Save
Reset to Default
Action Sets
Simple controller
HTC Vive wand
MS Motion controller
Touch controller
Output
Debugger
Audio
Animation
Shader Editor
OpenXR Action Map
4.0.beta

godot	Godot action set	0
trigger	Trigger	Float
trigger_click	Trigger click	Bool
trigger_touch	Trigger touching	Bool
grip	Grip	Float
grip_click	Grip click	Bool
grip_touch	Grip touching	Bool
primary	Primary joystick/thumbstick/trackpad	Vector2
primary_click	Primary joystick/thumbstick/trackpad click	Bool
primary_touch	Primary joystick/thumbstick/trackpad touching	Bool
secondary	Secondary joystick/thumbstick/trackpad	Vector2

Note

Godot's built-in input system has many things in common with the XR action map system. In fact our original idea was to add functionality to the existing input system and expose the data to the OpenXR action map system. We may revisit that idea at some point but as it turns out there were just too many problems to overcome. To name a few:

- Godot's input system mainly centers around button inputs, XR adds triggers, axis, poses and haptics (output) into the mix. This would greatly complicate the input system with features that won't work for normal controllers or contrast with the current approach. It was felt this would lead to confusion for the majority of Godot users.
- Godot's input system works with raw input data that is parsed and triggers emitting actions. This input data is made available to the end user. OpenXR completely hides raw data and does all the parsing for us, we only get

access to already parsed action data. This inconsistency is likely to lead to bugs when an unsuspecting user tries to use an XR device as a normal input device.

- Godot's input system allows changes to what inputs are bound to actions in runtime, OpenXR does not.
- Godot's input system is based on device ids which are meaningless in OpenXR.

This does mean that a game/application that mixes traditional inputs with XR controllers will have a separation. For most applications either one or the other is used and this is not seen as a problem. In the end, it's a limitation of the system.

The default action map

Godot will automatically create a default action map if no action map file is found.

Warning

This default map was designed to help developers port their XR games/applications from Godot 3 to Godot 4. As a result this map essentially binds all known inputs on all controllers supported by default, to actions one on one. This is not a good example of setting up an action map. It does allow a new developer to have a starting point when they want to become familiar with Godot XR. It prevents having to design a proper action map for their game/application first.

For this walkthrough we're going to start with a blank action map. You can simply delete the "Godot action set" entry at the top by pressing the trash can icon. This will clear out all actions. You might also want to remove the controllers that you do not wish to setup, more on this later.

Action sets

Note

Before we dive in, you will see the term XR runtime used throughout this document. With XR runtime we mean the software that is controlling and interacting with the AR or VR headset. The XR runtime then exposes this to us through an API such as OpenXR. So:

- for Steam this is SteamVR,
- for Meta on desktop this is the Oculus Client (including when using Quest link),
- for Meta on Quest this is the Quest's native OpenXR client,
- on Linux this could be Monado, etc.

The action map allows us to organize our actions in sets. Each set can be enabled or disabled on its own.

The concept here is that you could have different sets that provide bindings in different scenarios. You could have:

- a `Character control` set for when you're walking around,
- a `Vehicle control` set for when you're operating a vehicle,
- a `Menu` set for when a menu is open.

Only the action set applicable to the current state of your game/application can then be enabled.

This is especially important if you wish to bind the same input on a controller to a different action. For instance:

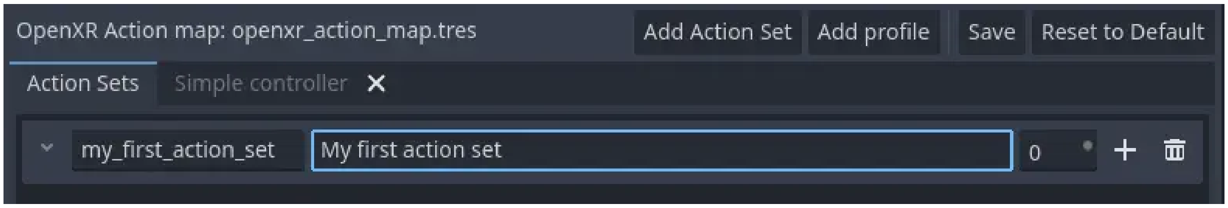
- in your `Character control` set you may have an action `Jump`,
- in your `Vehicle control` set you may have an action `Accelerate`,
- in your `Menu` set you may have an action `Select`.

All are bound to the trigger on your controller.

OpenXR will only bind an input or output to a single action. If the same input or output is bound to multiple actions the one in the active action set with the highest priority will be the one updated/used. So in our above example it will thus be important that only one action set is active.

For your first XR game/application we highly recommend starting with just a single action set and to not over-engineer things.

For our walkthrough in this document we will thus create a single action set called `my_first_action_set`. We do this by pressing the `Add action set` button:



my_first_action_set

My first action set

0

The columns in our table are as follows:

Col	Value	Description
1	my_first_action_set	This is the internal name of the action set. OpenXR doesn't specify specific restrictions on this name other then size,

Col	Value	Description
		however some XR runtimes will not like spaces or special characters.
2	My first action set	This is a human-readable name for the action set. Some XR runtimes will display this name to the end user, for example in configuration dialogs.
3	0	This is the priority of the action set. If multiple active action sets have actions bound to the same controllers inputs or outputs, the action set with the highest priority value will determine the action that is updated.

Actions

In the XR action map, actions are the entities that your game/application will interact with. For instance, we can define an action `Shoot` and the input bound to that action will trigger the `button_pressed` signal on the relevant XRController3D node in your scene with `Shoot` as the `name` parameter of the signal.

You can also poll the current state of an action. XRController3D for instance has an `is_button_pressed` method.

Actions can be used for both input and output and each action has a type that defines its behavior.

- The `Bool` type is used for discrete input like buttons.

- The `Float` type is used for analogue input like triggers.

These two are special as they are the only ones that are interchangeable. OpenXR will handle conversions between `Bool` and `Float` inputs and actions. You can get the value of a `Float` type action by calling the method `get_float` on your XRController3D node. It emits the `input_float_changed` signal when changed.

Note

Where analogue inputs are queried as buttons a threshold is applied. This threshold is currently managed exclusively by the XR runtime. There are plans to extend Godot to provide some level of control over these thresholds in the future.

The `Vector2` type defines the input as an axis input. Touchpads, thumbsticks and similar inputs are exposed as vectors. You can get the value of a `Vector2` type action by calling the method `get_vector2` on your XRController3D node. It emits the `input_vector2_changed` signal when changed.

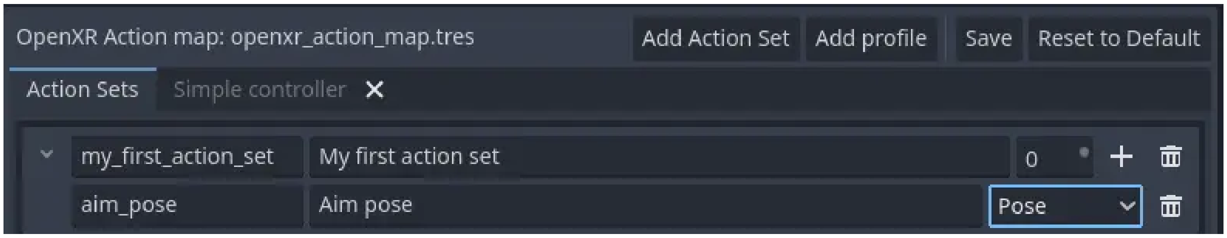
The `Pose` type defines a spatially tracked input. Multiple "pose" inputs are available in OpenXR: `aim`, `grip` and `palm`. Your XRController3D node is automatically positioned based on the pose action assigned to `pose` property of this node. More about poses later.

Note

The OpenXR implementation in Godot also exposes a special pose called `Skeleton`. This is part of the hand tracking implementation. This pose is exposed through the `skeleton` action that is supported outside of the action map system. It is thus always present if hand tracking is supported. You don't need to bind actions to this pose to use it.

Finally, the only output type is `Haptic` and it allows us to set the intensity of haptic feedback, such as controller vibration. Controllers can have multiple haptic outputs and support for haptic vests is coming to OpenXR.

So lets add an action for our aim pose, we do this by clicking on the `+` button for our action set:


OpenXR Action map: openxr_action_map.tres

my_first_action_set | My first action set | 0

aim_pose | Aim pose | Pose

The columns in our table are as follows:

Col	Value	Description
1	aim_pose	This is the internal name of the action. OpenXR doesn't specify specific restrictions on this name other then size, however some XR runtimes will not like spaces or special characters.
2	Aim pose	This is a human-readable name for the action. Some XR runtimes will display this name to the end user, for example in configuration dialogs.

Col	Value	Description
3	Pose	The type of this action.

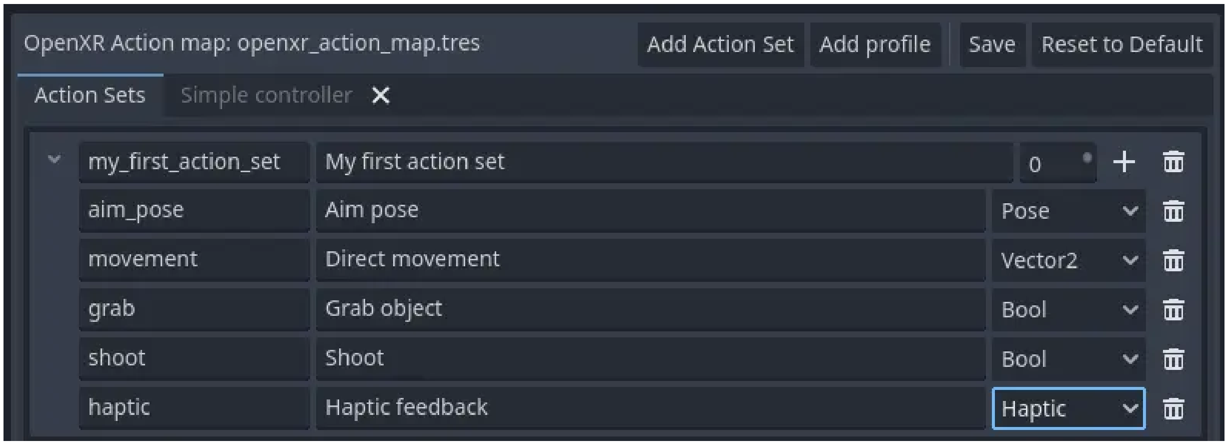
OpenXR defines a number of bindable input poses that are commonly available for controllers. There are no rules for which poses are supported for different controllers. The poses OpenXR currently defines are:

- The aim pose on most controllers is positioned slightly in front of the controller and aims forward. This is a great pose to use for laser pointers or to align the muzzle of a weapon with.
- The grip pose on most controllers is positioned where the grip button is placed on the controller. The orientation of this pose differs between controllers and can differ for the same controller on different XR runtimes.
- The palm pose on most controllers is positioned in the center of the palm of the hand holding the controller. This is a new pose that is not available on all XR runtimes.

Note

If hand tracking is used, there are currently big differences in implementations between the different XR runtimes. As a result the action map is currently not suitable for hand tracking. Work is being done on this so stay tuned.

Let's complete our list of actions for a very simple shooting game/application:


OpenXR Action map: openxr_action_map.tres
Add Action Set
Add profile
Save
Reset to Default
Action Sets
Simple controller

my_first_action_set	My first action set	0
aim_pose	Aim pose	Pose
movement	Direct movement	Vector2
grab	Grab object	Bool
shoot	Shoot	Bool
haptic	Haptic feedback	Haptic

The actions we have added are:

- movement, which allows the user to move around outside of normal room scale tracking.
- grab, which detects that the user wants to hold something.
- shoot, which detects that the user wants to fire the weapon they are holding.
- haptic, which allows us to output haptic feedback.

Now note that we don't distinguish between the left and right hand. This is something that is determined at the next stage. We've implemented the action system in such a way that you can bind the same action to both hands. The appropriate XRController3D node will emit the signal.

Warning

For both grab and shoot we've used the `Bool` type. As mentioned before, OpenXR does automatic conversions from an analogue controls however not all XR Runtimes currently apply sensible thresholds.

We recommend as a workaround to use the `Float` type when interacting with triggers and grip buttons and apply your own threshold.

For buttons like A/B/X/Y and similar where there is no analogue option, the `Bool` type works fine.

Note

You can bind the same action to multiple inputs for the same controller on the same profile. In this case the XR runtime will attempt to combine the inputs.

- For `Bool` inputs, this will perform an `OR` operation between the buttons.
- For `Float` inputs, this will take the highest value of the bound inputs.
- The behavior for `Pose` inputs is undefined, but the first bound input is likely to be used.

You shouldn't bind multiple actions of the same action set to the same controller input. If you do this, or if actions are bound from multiple action sets but they have overlapping priorities, the behavior is undefined. The XR runtime may simply not accept your action map, or it may take this on a first come first serve basis.

We are still investigating the restrictions around binding multiple actions to the same output as this scenario makes sense. The OpenXR specification seems to not allow this.

Now that we have our basic actions defined, it's time to hook them up.

Profiles

In OpenXR controller bindings are captured in so-called "Interaction Profiles". We've simply shortened it to "Profiles" because it takes up less space.

This generic name is chosen because controllers don't cover the entire system. Currently there are also profiles for trackers, remotes and tracked pens. There are also provisions for devices such as treadmills, haptic vests and such even though those are not part of the specification yet.

Warning

It is important to know that OpenXR has strict checking on supported devices. The core specification identifies a number of controllers and similar devices with their supported inputs and outputs. Every XR runtime must accept these interaction profiles even if they aren't applicable.

New devices are added through extensions and XR runtimes must specify which ones they support. XR runtimes that do not support a device added through extensions will not accept these profiles. XR runtimes that do not support added input or output types will often crash if supplied.

As such Godot keeps meta data of all available devices, their inputs and outputs and which extension adds support for them. You can create interaction profiles for all devices you wish to support. Godot will filter out those not supported by the XR runtime the user is using.

This does mean that in order to support new devices, you might need to update to a more recent version of Godot.

It is however also important to note that the action map has been designed with this in mind. When new devices enter the market, or when your users use devices that you do not have access to, the action map system relies on the XR runtime. It is the XR runtime's job to choose the best fitting interaction profile that has been specified and adapt it for the controller the user is using.

How the XR runtime does this is left to the implementation of the runtime and there are thus vast differences between the runtimes. Some runtimes might even permit users to edit the bindings themselves.

A common approach for a runtime is to look for a matching interaction profile first. If this is not found it will check the most common profiles such as that of the "Touch controller" and do a conversion. If all else fails, it will check the generic "Simple controller".

Note

There is an important conclusion to be made here: When a controller is found, and the action map is applied to it, the XR runtime is not limited to the exact configurations you set up in Godot's action map editor. While the runtime will generally choose a suitable mapping based on one of the bindings you set up in the action map, it can deviate from it.

For example, when the Touch controller profile is used any of the following scenarios could be true:

- we could be using a Quest 1 controller,
- we could be using a Quest 2 controller,
- we could be using a Quest Pro controller but no Quest Pro profile was given or the XR runtime being used does not support the Quest Pro controller,
- it could be a completely different controller for which no profile was given but the XR runtime is using the touch bindings as a base.

Ergo, there currently is no way to know with certainty, which controller the user is actually using.

Warning

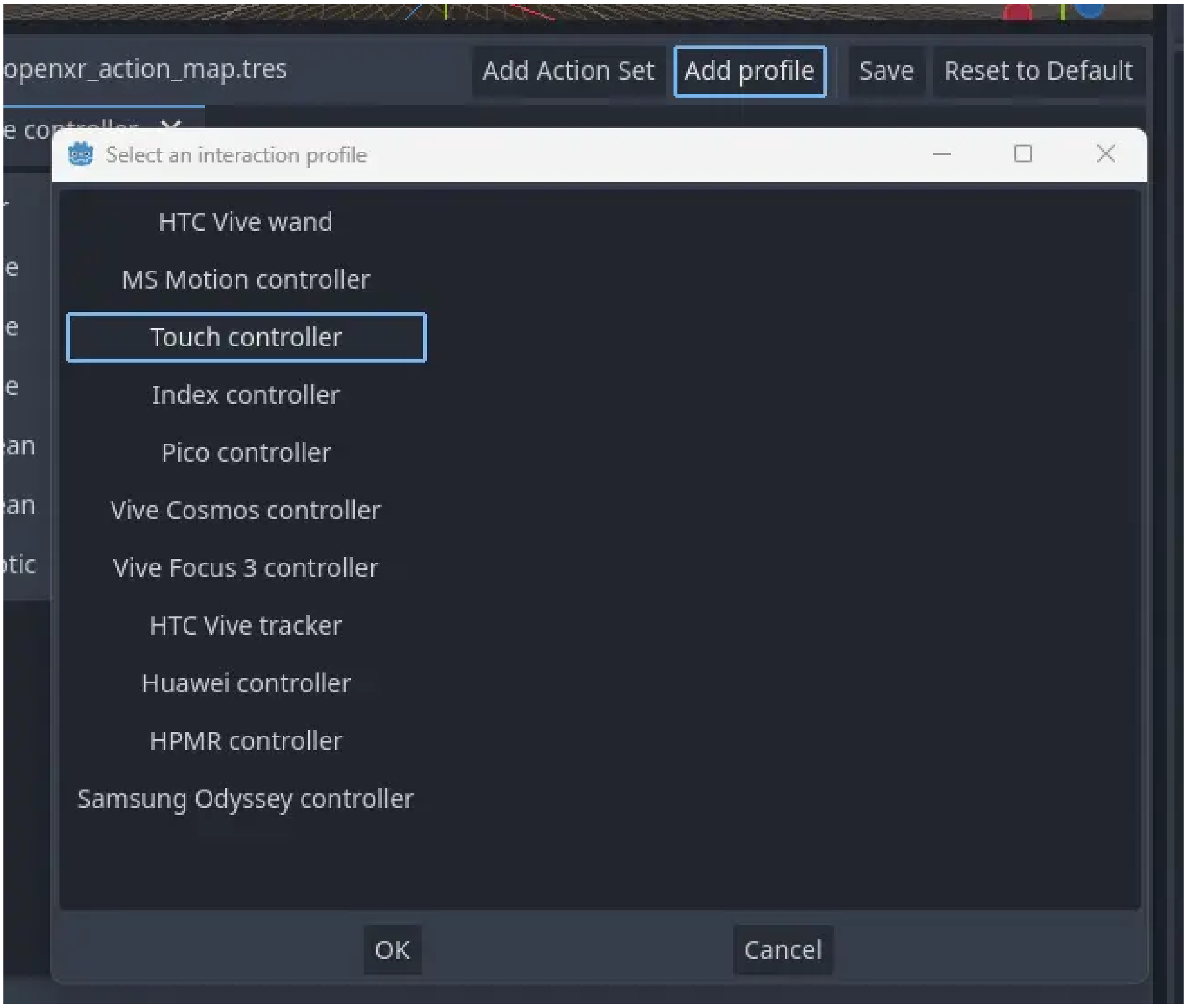
Finally, and this trips up a lot of people, the bindings aren't set in stone. It is fully allowed, and even expected, that an XR runtime allows a user to customise the bindings.

At the moment none of the XR runtimes offer this functionality though SteamVR has an existing UI from OpenVRs action map system that is still accessible. This is actively being worked on however.

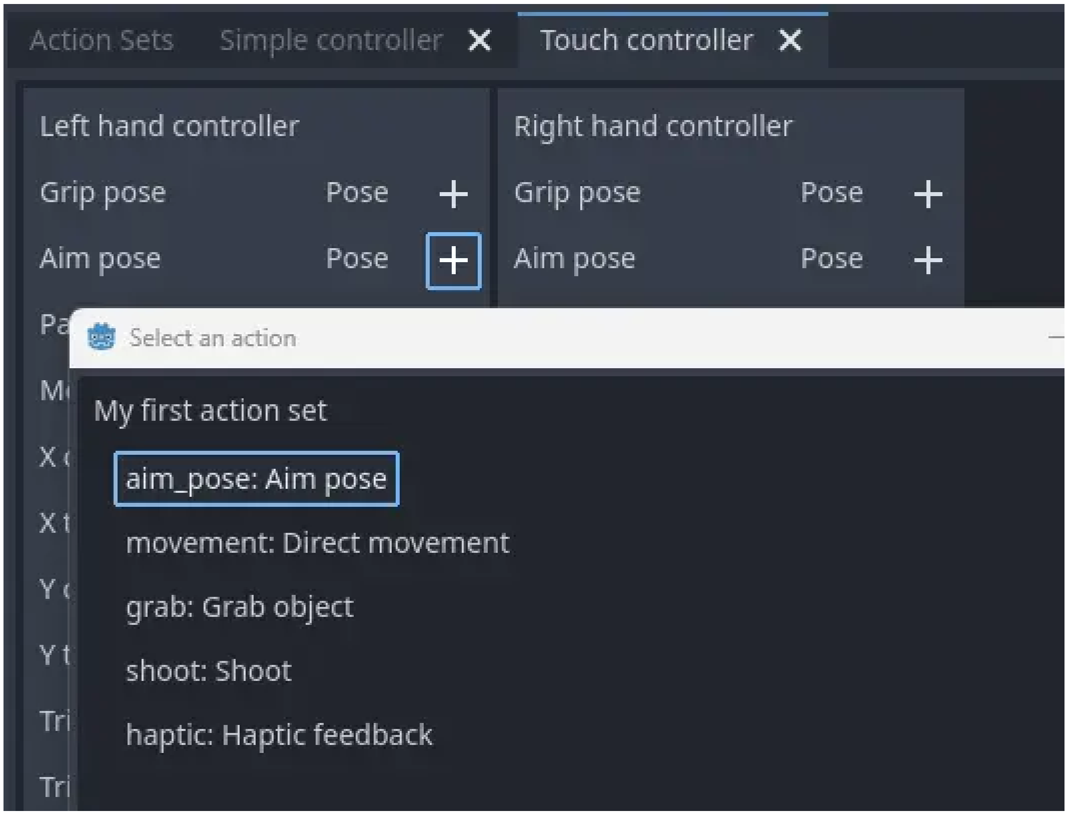
Our first controller binding

Let's set up our first controller binding, using the Touch controller as an example.

Press "Add profile", find the Touch controller, and add it. If it is not in the list, then it may already have been added.


openxr_action_map.tres
Add Action Set
Add profile
Save
Reset to Default
Select an interaction profile
HTC Vive wand
MS Motion controller
Touch controller
Index controller
Pico controller
Vive Cosmos controller
Vive Focus 3 controller
HTC Vive tracker
Huawei controller
HPMR controller
Samsung Odyssey controller
OK
Cancel

Our UI now shows panels for both the left and right controllers. The panels contain all of the possible inputs and outputs for each controller. We can use the + next to each entry to bind it to an action:

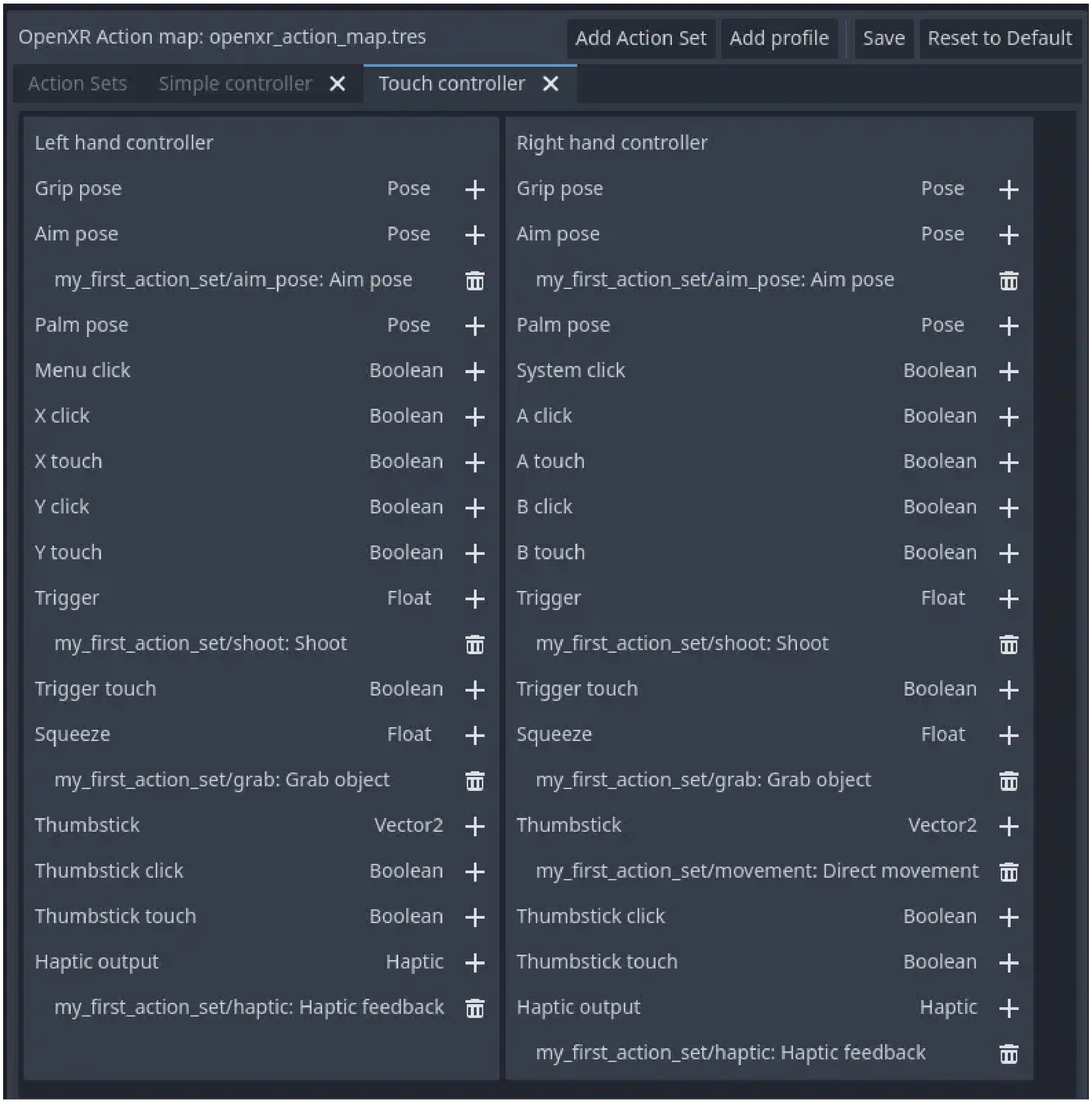


Action Sets
Simple controller
Touch controller
Left hand controller
Grip pose
Pose
Aim pose
Pose
Right hand controller
Grip pose
Pose
Aim pose
Pose
Select an action

My first action set

- aim_pose: Aim pose
- movement: Direct movement
- grab: Grab object
- shoot: Shoot
- haptic: Haptic feedback

Let's finish our configuration:



Left hand controller

Input	Type
Grip pose	Pose
Aim pose	Pose
my_first_action_set/aim_pose: Aim pose	
Palm pose	Pose
Menu click	Boolean
X click	Boolean
X touch	Boolean
Y click	Boolean
Y touch	Boolean
Trigger	Float
my_first_action_set/shoot: Shoot	
Trigger touch	Boolean
Squeeze	Float
my_first_action_set/grab: Grab object	
Thumbstick	Vector2
Thumbstick click	Boolean
Thumbstick touch	Boolean
Haptic output	Haptic
my_first_action_set/haptic: Haptic feedback	

Right hand controller

Input	Type
Grip pose	Pose
Aim pose	Pose
my_first_action_set/aim_pose: Aim pose	
Palm pose	Pose
System click	Boolean
A click	Boolean
A touch	Boolean
B click	Boolean
B touch	Boolean
Trigger	Float
my_first_action_set/shoot: Shoot	
Trigger touch	Boolean
Squeeze	Float
my_first_action_set/grab: Grab object	
Thumbstick	Vector2
my_first_action_set/movement: Direct movement	
Thumbstick click	Boolean
Thumbstick touch	Boolean
Haptic output	Haptic
my_first_action_set/haptic: Haptic feedback	

Each action is bound the given input or output for both controllers to indicate that we support the action on either controller. The exception is the movement action which is bound only to the right hand controller. It is likely that we would want to use the left hand thumbstick for a different purpose, say a teleport function.

In developing your game/application you have to account for the possibility that the user changes the binding and binds the movement to the left hand thumbstick.

Also note that our shoot and grab boolean actions are linked to inputs of type `Float`. As mentioned before OpenXR will do conversions between the two, but do read the warning given on that subject earlier in this document.

Note

Some of the inputs seem to appear in our list multiple times.

For instance we can find the `X` button twice, once as `X click` and then as `X touch`. This is due to the Touch controller having a capacitive sensor.

- `X touch` will be true if the user is merely touching the X button.
- `X click` will be true when the user is actually pressing down on the button.

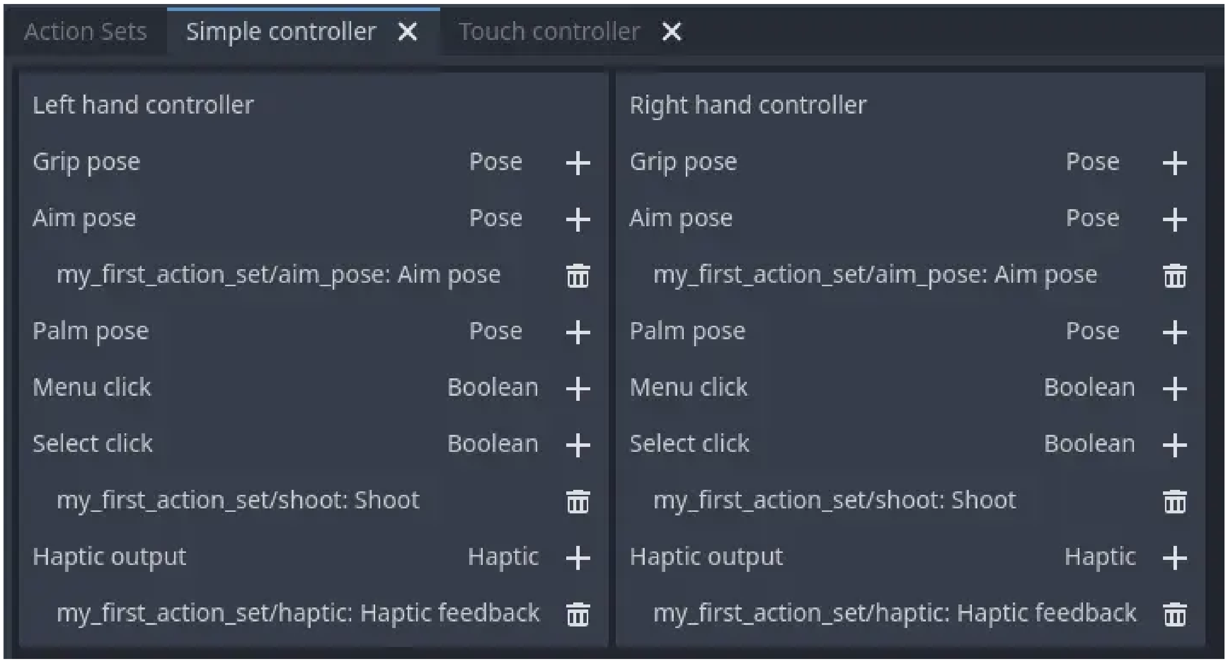
Similarly for the thumbstick we have:

- `Thumbstick touch` which will be true if the user is touching the thumbstick.
- `Thumbstick` which gives a value for the direction the thumbstick is pushed to.
- `Thumbstick click` which is true when the user is pressing down on the thumbstick.

It is important to note that only a select number of XR controllers support touch sensors or have click features on thumbsticks. Keep that in mind when designing your game/application. Make sure these are used for optional features of your game/application.

The simple controller

The "Simple controller" is a generic controller that OpenXR offers as a fallback. We'll apply our mapping:



Left hand controller		Right hand controller	
Grip pose	Pose	Grip pose	Pose
Aim pose	Pose	Aim pose	Pose
my_first_action_set/aim_pose: Aim pose		my_first_action_set/aim_pose: Aim pose	
Palm pose	Pose	Palm pose	Pose
Menu click	Boolean	Menu click	Boolean
Select click	Boolean	Select click	Boolean
my_first_action_set/shoot: Shoot		my_first_action_set/shoot: Shoot	
Haptic output	Haptic	Haptic output	Haptic
my_first_action_set/haptic: Haptic feedback		my_first_action_set/haptic: Haptic feedback	

As becomes painfully clear, the simple controller is often far too simple and falls short for anything but the simplest of VR games/applications.

This is why many XR runtimes only use it as a last resort and will attempt to use bindings from one of the more popular systems as a fallback first.

Note

Due to the simple controller likely not covering the needs of your game, it is tempting to provide bindings for every controller supported by OpenXR. The default action map seems to suggest this as a valid course of action. As mentioned before, the default action map was designed for ease of migration from Godot 3.

It is the recommendation from the OpenXR Working Group that only bindings for controllers actually tested by the developer are setup. The XR runtimes are designed with this in mind. They can perform a better job of rebinding a provided binding than a developer can make educated guesses. Especially as the developer can't test if this leads to a comfortable experience for the end user.

This is our advice as well: limit your action map to the interaction profiles for devices you have actually tested your game with. The Oculus Touch controller is widely used as a fallback controller by many runtimes. If you are able to test your game using a Meta Rift or Quest and add this profile there is a high probability your game will work with other headsets.

Room scale in XR

One of the staples of XR projects is the ability to walk around freely in a large space. This space is often constrained by the room the player is physically in with tracking sensors placed within this space. With the advent of inside out tracking however ever larger play spaces are possible.

As a developer this introduces a number of interesting challenges. In this document we will look at a number of the challenges you may face and outline some solutions. We'll discuss the issues and challenges for seated XR games in another document.

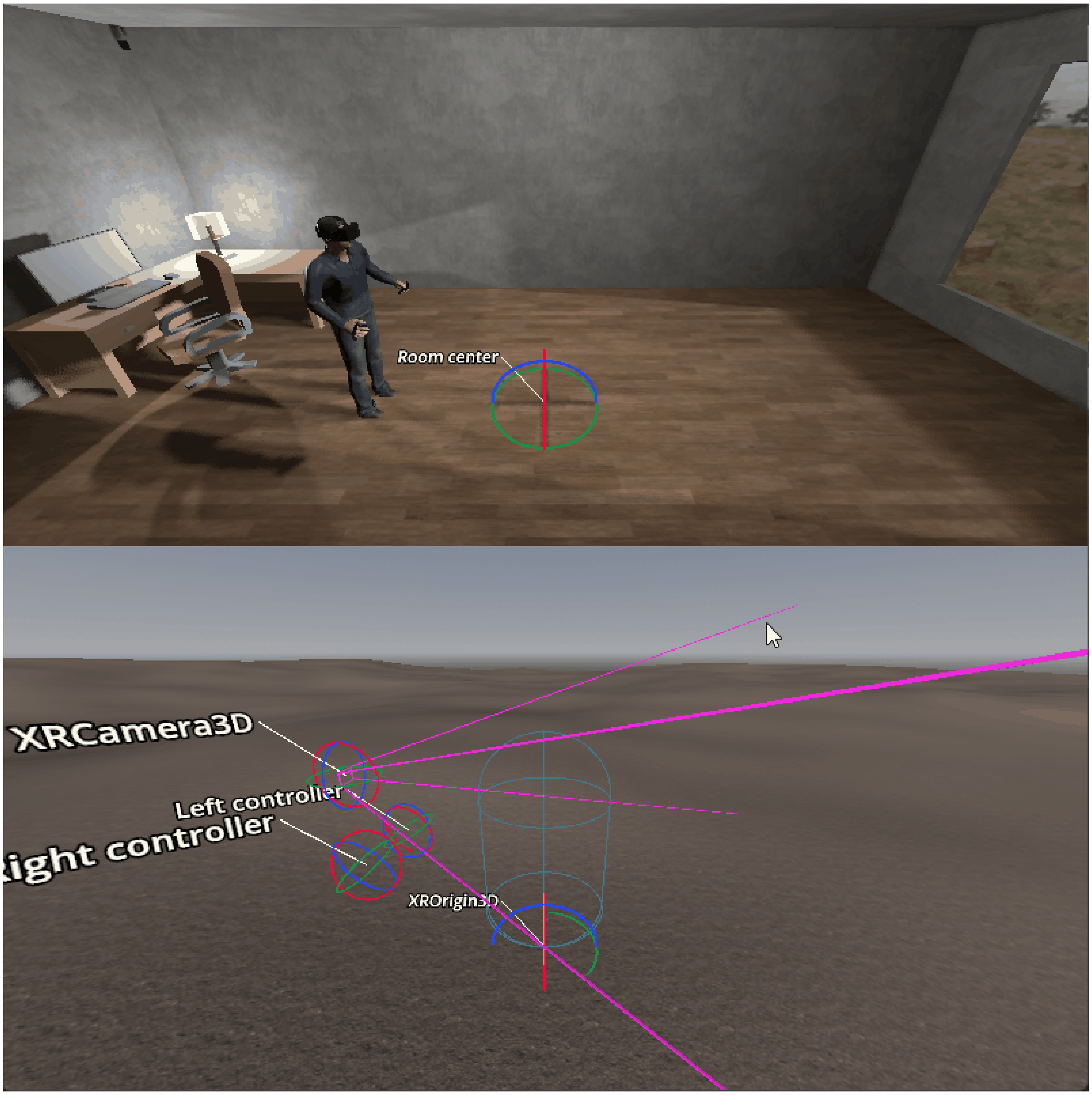
Note

Often developers sit behind their desk while building the foundation to their game. In this mode the issues with developing for room scale don't show themselves until it is too late. The advice here is to start testing while standing up and walking around as early as possible. Once you are happy your foundation is solid, you can develop in comfort while remaining seated.

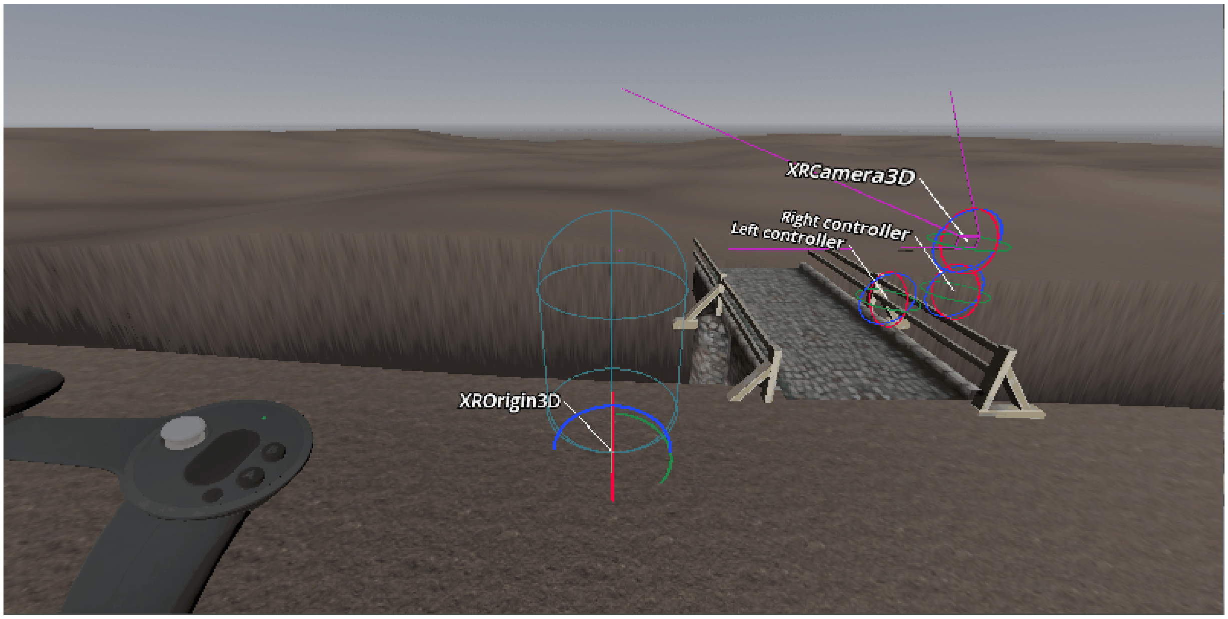
In traditional first person games a player is represented by a CharacterBody3D node. This node is moved by processing traditional controller, mouse or keyboard input. A camera is attached to this node at a location roughly where the player's head will be.

Applying this model to the XR setup, we add an XROrigin3D node as a child of the character body, and add an XRCamera3D as a child of the origin node. At face value this seems to work. However, upon closer examination this model does not take into account that there are two forms of movement in XR. The movement through controller input, and the physical movement of the player in the real world.

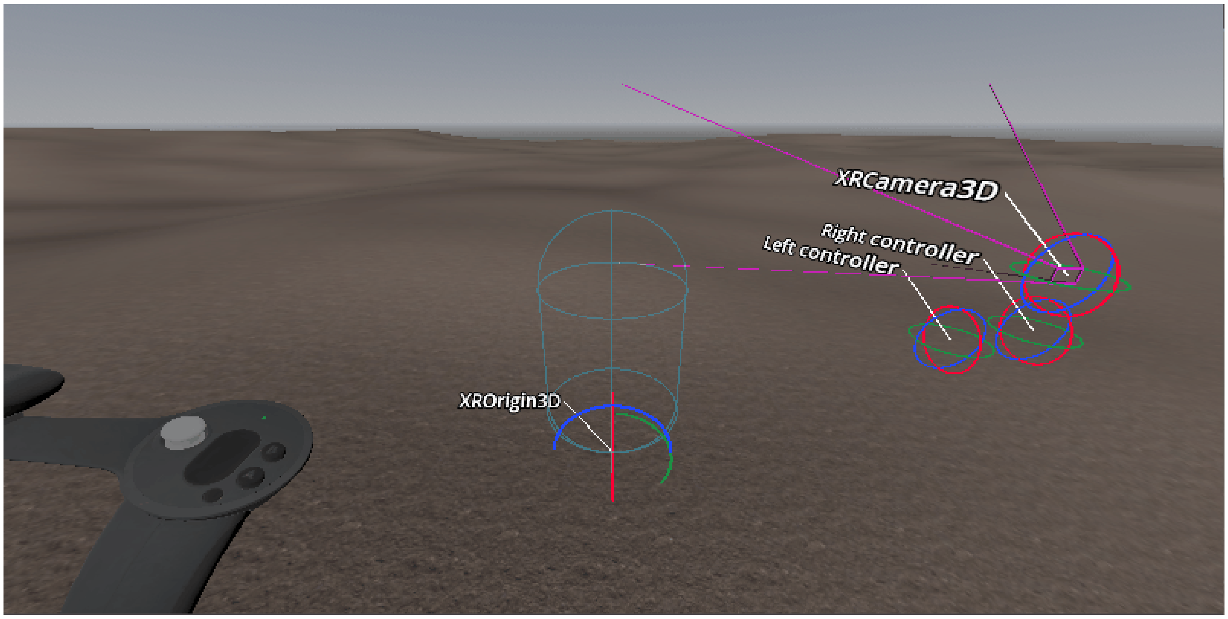
As a result, the origin node does not represent the position of the player. It represents the center, or start of, the tracking space in which the player can physically move. As the player moves around their room this movement is represented through the tracking of the players headset. In game this translates to the camera node's position being updated accordingly. For all intents and purposes, we are tracking a disembodied head. Unless body tracking is available, we have no knowledge of the position or orientation of the player's body.


Room center
XRCamera3D
Left controller
ight controller
XROrigin3D

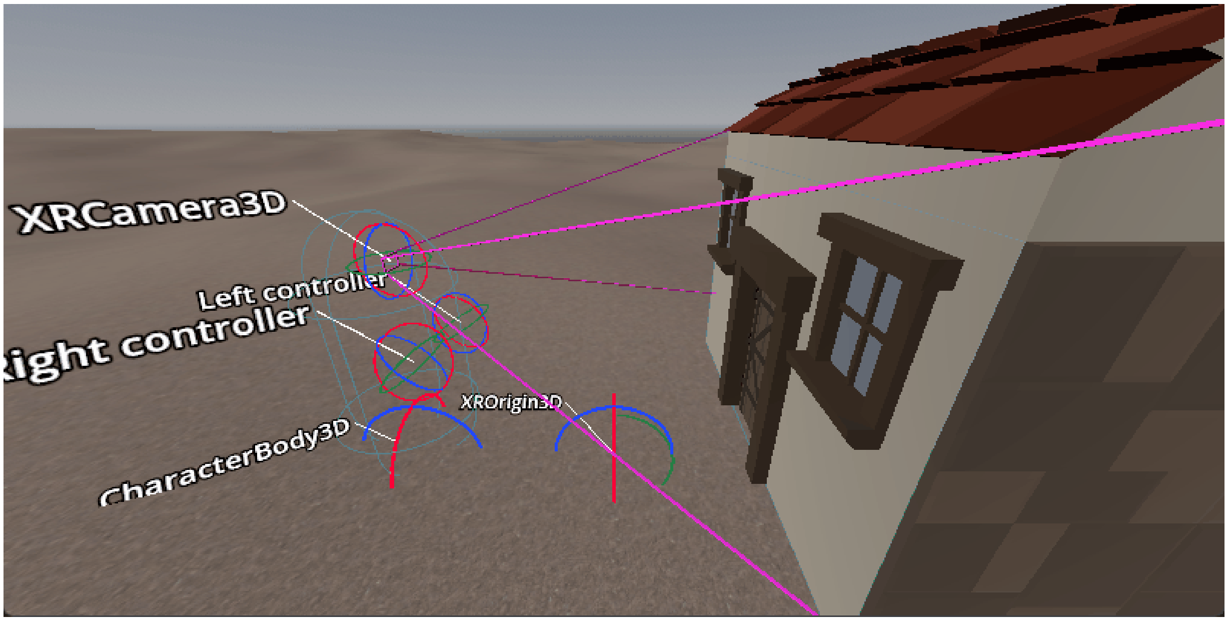
The first problem this causes is fairly obvious. When the player moves with controller input, we can use the same approach in normal games and move the player in a forward direction. However the player isn't where we think they are and as we move forward we're checking collisions in the wrong location.



The second problem really shows itself when the player walks further away from the center of the tracking space and uses controller input to turn. If we rotate our character body, the player will be moved around the room in a circular fashion.


XRCamera3D
Right controller
Left controller
XROrigin3D

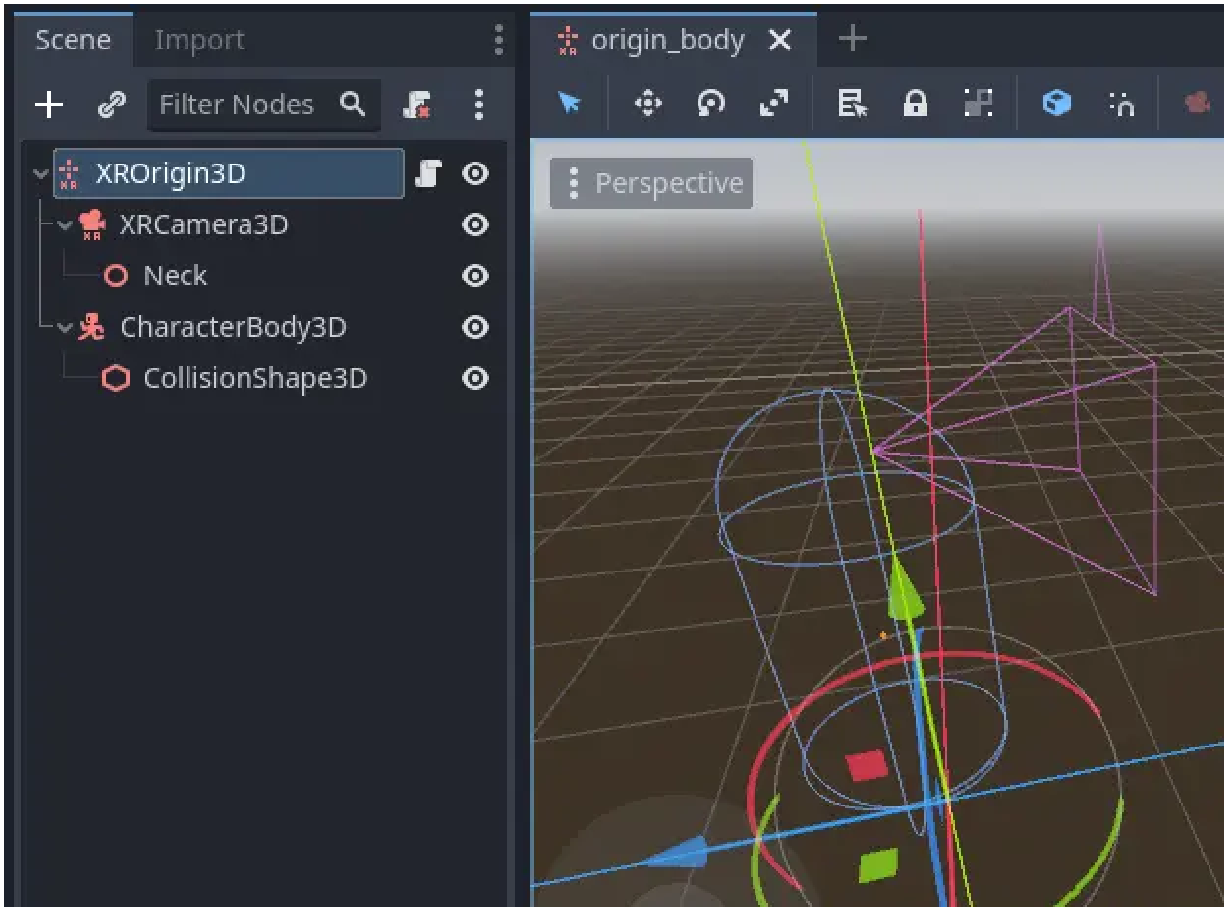
If we fix the above issues, we will find a third issue. When the path for the player is blocked in the virtual world, the player can still physically move forward.


XRCamera3D
Left controller
Right controller
CharacterBody3D
XROrigin3D

We will look at solving the first two problem with two separate solutions, and then discuss dealing with the third.

Origin centric solution

Looking at the first approach for solving this we are going to change our structure. This is the approach currently implemented in XR Tools.


Scene
Import
Filter Nodes
XROrigin3D
XRCamera3D
Neck
CharacterBody3D
CollisionShape3D
origin_body
Perspective

In this setup we mark the character body as top level so it does not move with the origin.

We also have a helper node that tells us where our neck joint is in relation to our camera. We use this to determine where our body center is.

Processing our character movement is now done in three steps.

Note

The Origin centric movement demo [https://github.com/godotengine/godot-demo-projects/tree/master/xr/openxr_origin_centric_movement] contains a more elaborate example of the technique described below.

Step 1

In the first step we're going to process the physical movement of the player. We determine where the player is right now, and attempt to move our character body there.

```
func _process_on_physical_movement(delta):
  # Remember our current velocity, we'll apply that later
  var current_velocity = $CharacterBody3D.velocity

  # Remember where our player body currently is
  var org_player_body: Vector3 = $CharacterBody3D.global_transform.origin

  # Determine where our player body should be
  var player_body_location: Vector3 = $XRCamera3D.transform * $XRCamera3D/Neck.transform.origin
  player_body_location.y = 0.0
  player_body_location = global_transform * player_body_location

  # Attempt to move our character
  $CharacterBody3D.velocity = (player_body_location - org_player_body) / delta
  $CharacterBody3D.move_and_slide()

  # Set back to our current value
  $CharacterBody3D.velocity = current_velocity

  # Check if we managed to move all the way, ignoring height change
  var movement_left = player_body_location -
$CharacterBody3D.global_transform.origin
  movement_left.y = 0.0
  if (movement_left).length() > 0.01:
    # We'll talk more about what we'll do here later on
    return true
  else:
    return false

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
```

Note that we're returning `true` from our `_process_on_physical_movement` function when we couldn't move our player all the way.

Step 2

The second step is to handle rotation of the player as a result of user input.

As the input used can differ based on your needs we are simply calling the function `_get_rotational_input`. This function should obtain the necessary input and return the rotational speed in radians per second.

Note

For our example we are going to keep this simple and straight forward. We are not going to worry about comfort features such as snap turning and applying a vignette. We highly recommend implementing such comfort features.

```
func _get_rotational_input() -> float:
  # Implement this function to return rotation in radians per
second.
```

```
  return 0.0

func _copy_player_rotation_to_character_body():
  # We only copy our forward direction to our character body,
we ignore tilt
  var camera_forward: Vector3 =
-$XRCamera3D.global_transform.basis.z
  var body_forward: Vector3 = Vector3(camera_forward.x, 0.0,
camera_forward.z)

  $CharacterBody3D.global_transform.basis =
Basis.looking_at(body_forward, Vector3.UP)

func _process_rotation_on_input(delta):
  var t1 := Transform3D()
  var t2 := Transform3D()
  var rot := Transform3D()

  # We are going to rotate the origin around the player
  var player_position =
$CharacterBody3D.global_transform.origin -
global_transform.origin

  t1.origin = -player_position
  t2.origin = player_position
  rot = rot.rotated(Vector3(0.0, 1.0, 0.0),
_get_rotational_input() * delta)
  global_transform = (global_transform * t2 * rot *
t1).orthonormalized()

  # Now ensure our player body is facing the correct way as
well
  _copy_player_rotation_to_character_body()

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
  if !is_colliding:
    _process_rotation_on_input(delta)
```

Note

We've added the call for processing our rotation to our physics process but we are only executing this if we were able to move our player fully. This means that if the player moves somewhere they shouldn't, we don't process further movement.

Step 3

The third and final step is moving the player forwards, backwards or sideways as a result of user input.

Just like with the rotation the inputs differ from project to project so we are simply calling the function `_get_movement_input`. This function should obtain the necessary input and return a directional vector scaled to the required velocity.

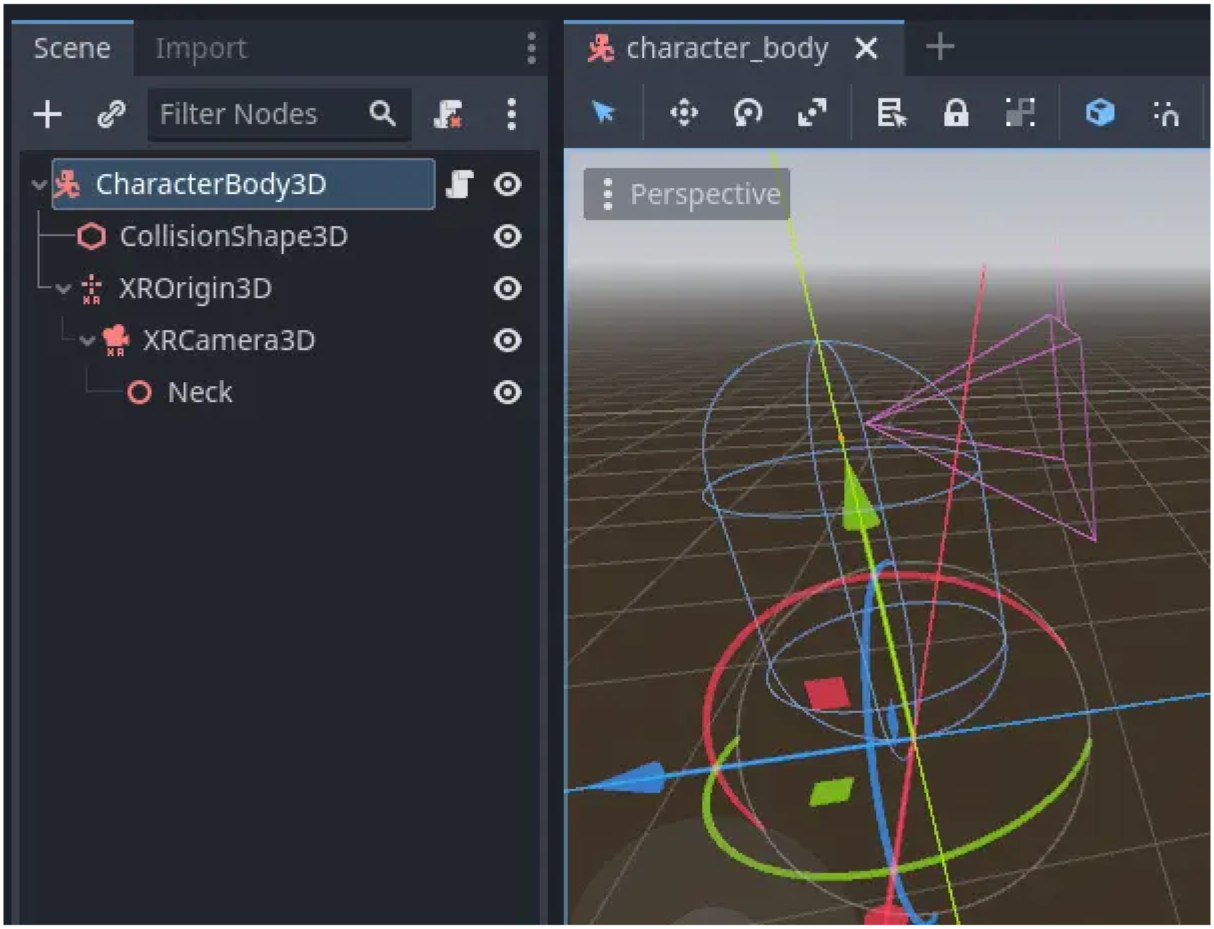
Note

Just like with rotation we're keeping it simple. Here too it is advisable to look at adding comfort settings.

```
var gravity =
ProjectSettings.get_setting("physics/3d/default_gravity")

func _get_movement_input() -> Vector2:
  # Implement this to return requested directional movement in
meters per second.
  return Vector2()

func _process_movement_on_input(delta):
  # Remember where our player body currently is
  var org_player_body: Vector3 =
$CharacterBody3D.global_transform.origin

  # We start with applying gravity
  $CharacterBody3D.velocity.y -= gravity * delta
  # Now we add in our movement
  var input: Vector2 = _get_movement_input()
  var movement: Vector3 =
($CharacterBody3D.global_transform.basis * Vector3(input.x, 0,
input.y))
  $CharacterBody3D.velocity.x = movement.x
  $CharacterBody3D.velocity.z = movement.z

  # Attempt to move our player
  $CharacterBody3D.move_and_slide()

  # And now apply the actual movement to our origin
  global_transform.origin +=
$CharacterBody3D.global_transform.origin - org_player_body

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
  if !is_colliding:
    _process_rotation_on_input(delta)
    _process_movement_on_input(delta)
```

Character body centric solution

In this setup we are going to keep our character body as our root node and as such is easier to combine with traditional game mechanics.


Scene
Import
Filter Nodes
CharacterBody3D
CollisionShape3D
XROrigin3D
XRCamera3D
Neck
character_body
Perspective

Here we have a standard character body with collision shape, and our XR origin node and camera as normal children. We also have our neck helper node.

Processing our character movement is done in the same three steps but implemented slightly differently.

Note

The Character centric movement demo [https://github.com/godotengine/godot-demo-projects/tree/master/xr/openxr_character_centric_movement] contains a more elaborate example of the technique described below.

Step 1

In this approach step 1 is where all the magic happens. Just like with our previous approach we will be applying our physical movement to the character body, but we will counter that movement on the origin node.

This will ensure that the players location stays in sync with the character body's location.

```
# Helper variables to keep our code readable
@onready var origin_node = $XROrigin3D
@onready var camera_node = $XROrigin3D/XRCamera3D
@onready var neck_position_node = $XROrigin3D/XRCamera3D/Neck

func _process_on_physical_movement(delta) -> bool:
  # Remember our current velocity, we'll apply that later
  var current_velocity = velocity

  # Start by rotating the player to face the same way our real player is
  var camera_basis: Basis = origin_node.transform.basis * camera_node.transform.basis
  var forward: Vector2 = Vector2(camera_basis.z.x, camera_basis.z.z)
  var angle: float = forward.angle_to(Vector2(0.0, 1.0))

  # Rotate our character body
  transform.basis = transform.basis.rotated(Vector3.UP, angle)

  # Reverse this rotation our origin node
  origin_node.transform = Transform3D().rotated(Vector3.UP, -angle) * origin_node.transform

  # Now apply movement, first move our player body to the right location
  var org_player_body: Vector3 = global_transform.origin
  var player_body_location: Vector3 = origin_node.transform * camera_node.transform * neck_position_node.transform.origin
  player_body_location.y = 0.0
  player_body_location = global_transform *
player_body_location

  velocity = (player_body_location - org_player_body) / delta
  move_and_slide()

  # Now move our XROrigin back
  var delta_movement = global_transform.origin -
org_player_body
  origin_node.global_transform.origin -= delta_movement

  # Return our value
  velocity = current_velocity

  if (player_body_location - global_transform.origin).length()
> 0.01:
    # We'll talk more about what we'll do here later on
    return true
  else:
    return false

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
```

In essence the code above will move the character body to where the player is, and then move the origin node back in equal amounts. The result is that the player stays centered above the character body.

We start with applying the rotation. The character body should be facing where the player was looking the previous frame. We calculate our camera orientation in the space of the character body. We can now calculate the angle by which the player has rotated their head. We rotate our character body by the same amount so our character body faces the same direction as the player. And then we reverse the rotation on the origin node so the camera ends up aligned with the player again.

For the movement we do much the same. The character body should be where the player was standing the previous frame. We calculate by how much the player has moved from this location. Then we attempt to move the character body to this location.

As the player may hit a collision body and be stopped, we only move the origin point back by the amount we actually moved the character body. The player may thus move away from this location but that will be reflected in the positioning of the player.

As with our previous solution we return true if this is the case.

Step 2

In this step we again apply the rotation based on controller input. However in this case the code is nearly identical to how one would implement this in a normal first person game.

As the input used can differ based on your needs we are simply calling the function `_get_rotational_input`. This function should obtain the necessary input and return the rotational speed in radians per second.

```
func _get_rotational_input() -> float:
  # Implement this function to return rotation in radians per
second.
  return 0.0

func _process_rotation_on_input(delta):
  rotation.y += _get_rotational_input() * delta

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
  if !is_colliding:
    _process_rotation_on_input(delta)
```

Step 3

For step three we again apply the movement based on controller input. However just like at step 2, we can now implement this as we would in a normal first person game.

Just like with the rotation the inputs differ from project to project so we are simply calling the function `_get_movement_input`. This function should obtain the necessary input and return a directional vector scaled to the required velocity.

```
func _get_movement_input() -> Vector2:
  # Implement this to return requested directional movement in
meters per second.
  return Vector2()

func _process_movement_on_input(delta):
  var movement_input = _get_movement_input()
  var direction = global_transform.basis *
Vector3(movement_input.x, 0, movement_input.y)
  if direction:
    velocity.x = direction.x
    velocity.z = direction.z
  else:
    velocity.x = move_toward(velocity.x, 0, delta)
    velocity.z = move_toward(velocity.z, 0, delta)

  move_and_slide()

func _physics_process(delta):
  var is_colliding = _process_on_physical_movement(delta)
  if !is_colliding:
    _process_rotation_on_input(delta)
    _process_movement_on_input(delta)
```

When the player walks to somewhere they shouldn't

Think of a situation where the player is outside a locked room. You don't want the player to go into that room until the door is unlocked. You also don't want the player to see what is in this room.

The logic for moving the player through controller input nicely prevents this. The player encounters a static body, and the code prevents the player from moving into the room.

However with XR, nothing is preventing the player from taking a real step forward.

With both the approaches worked out up above we will prevent the character body from moving where the player can't go. As the player has physically moved to this location, the camera will now have moved into the room.

The logical solution would be to prevent the movement altogether and adjust the placement of the XR origin point so the player stays outside of the room.

The problem with this approach is that physical movement is now not replicated in the virtual space. This will cause nausea for the player.

What many XR games do instead, is to measure the distance between where the player physically is, and where the players virtual body has been left behind. As this distance increases, usually to a distance of a few centimeters, the screen slowly blacks out.

Our solutions up above would allow us to add this logic into the code at the end of step 1.

Further improvements to the code presented could be:

- allowing controller input as long as this distance is still small,

- still applying gravity to the player even when controller input is disabled.

Note

The movement demos in our demo repository contain an example of blacking out the screen when a user walks into restricted areas.

Further suggestions for improvements

The above provides two good options as starting points for implementing room scale XR games.

A few more things that are worth pointing out that you will likely want to implement:

- The height of the camera can be used to detect whether the player is standing up, crouching, jumping or lying down. You can adjust the size and orientation of the collision shape accordingly. Extra bonus points for adding multiple collision shapes so the head and body have their own, more accurately sized, shapes.
- When a scene first loads, the player may be far away from the center of the tracking space. This could result in the player spawning into a different room than our origin point. The game will now attempt, and fail, to move the player body from the starting point to where the player is standing. You should implement a reset function that moves the origin point so the player is in the correct starting position.

Both of the above improvements require the player to be ready and standing up straight. There is no guarantee as the player may still be putting their headset on.

Many games, including XR Tools, solve this by introducing an intro screen or loading screen where the player must press a button when they are ready. This starting environment is often a large location where the positioning of the player has little impact on what the player sees. When the player is ready, and presses the button, this is the moment you record the position and height of the camera.

OpenXR composition layers

Introduction

In XR games you generally want to create user interactions that happen in 3D space and involve users touching objects as if they are touching them in real life.

Sometimes however creating a more traditional 2D interface is unavoidable. In XR however you can't just add 2D components to your scene. Godot needs depth information to properly position these elements so they appear at a comfortable place for the user. Even with depth information there are headsets with slanted displays that make it impossible for the standard 2D pipeline to correctly render the 2D elements.

The solution then is to render the UI to a SubViewport and display the result of this using a ViewportTexture on a 3D mesh. The QuadMesh is a suitable option for this.

Note

See the GUI in 3D [https://github.com/godotengine/godot-demo-projects/tree/master/viewport/gui_in_3d] example project for an example of this approach.

The problem with displaying the viewport in this way is that the rendered result is sampled for lens distortion by the XR runtime and the resulting quality loss can make UI text hard to read.

OpenXR offers a solution to this problem through composition layers. With composition layers it is possible for the contents of a viewport to be projected on a surface after lens distortion resulting in a much higher quality end result.

Note

As not all XR runtimes support all composition layer types, Godot implements a fallback solution where we render the viewport as part of the normal scene but with the aforementioned quality limitations.

Warning

When the composition layer is supported, it is the XR runtime that presents the subviewport. This means the UI is only visible in the headset, it will not be accessible by Godot and will thus not be shown when you have a spectator view on the desktop.

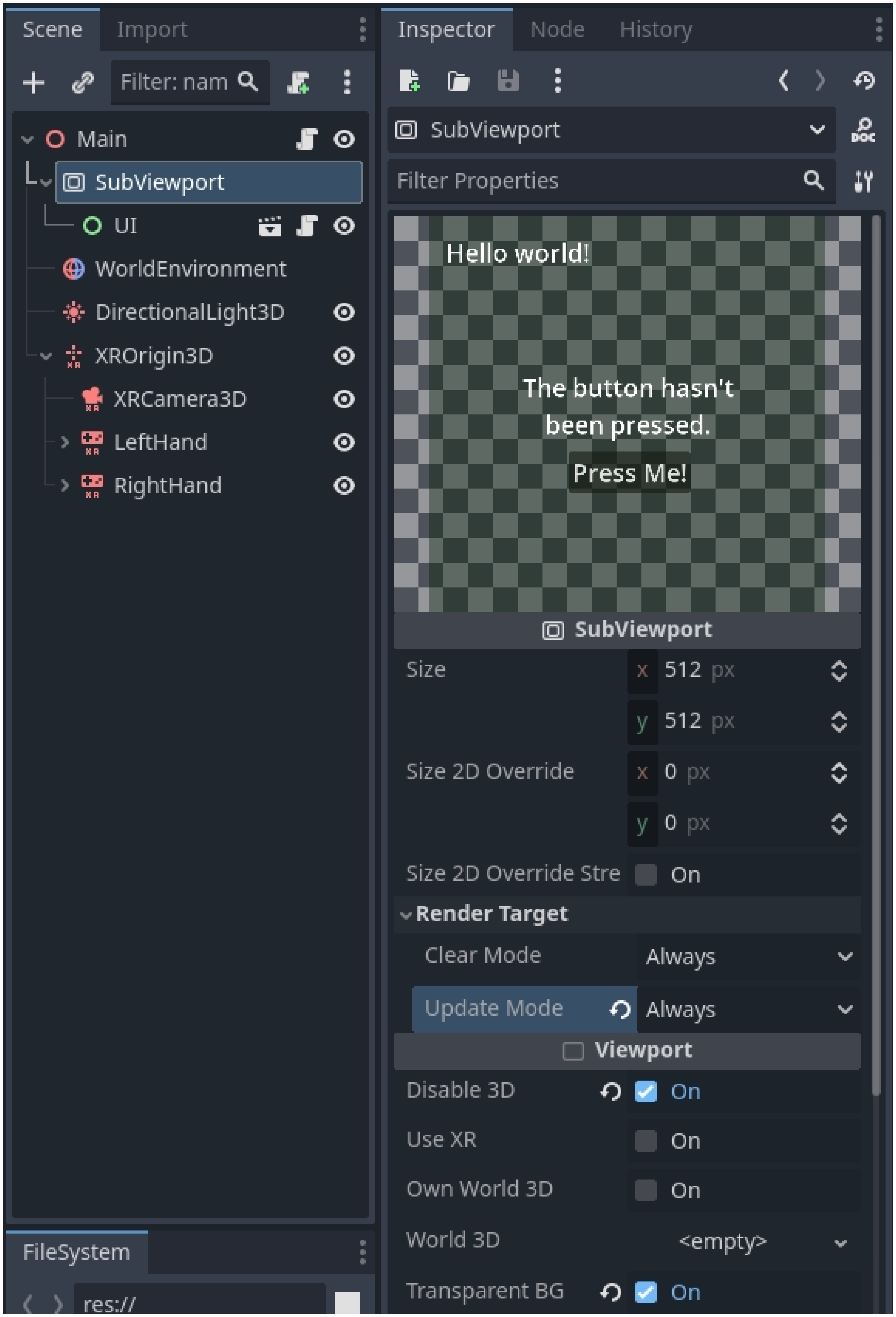
There are currently 3 nodes that expose this functionality:

- OpenXRCompositionLayerCylinder shows the contents of the SubViewport on the inside of a cylinder (or "slice" of a cylinder).
- OpenXRCompositionLayerEquirect shows the contents of the SubViewport on the interior of a sphere (or "slice" of a sphere).
- OpenXRCompositionLayerQuad shows the contents of the SubViewport on a flat rectangle.

Setting up the SubViewport

The first step is adding a SubViewport for our 2D UI, this doesn't require any specific steps. For our example we do mark the viewport as transparent.

You can now create the 2D UI by adding child nodes to the SubViewport as you normally would. It is advisable to save the 2D UI in a subscene, this makes it easier to do your layout.



Scene
Import
Filter: nam
Main
SubViewport
UI
WorldEnvironment
DirectionalLight3D
XROrigin3D
XRCamera3D
LeftHand
RightHand
FileSystem
res://
Inspector
Node
History
SubViewport
Filter Properties
Hello world!
The button hasn't been pressed.
Press Me!
SubViewport

Size x 512 px
y 512 px

Size 2D Override x 0 px
y 0 px

Size 2D Override Stre ☐ On

Render Target

Clear Mode Always

Update Mode Always

Viewport

Disable 3D ☑ On

Use XR ☐ On

Own World 3D ☐ On

World 3D <empty>

Transparent BG ☑ On

Warning

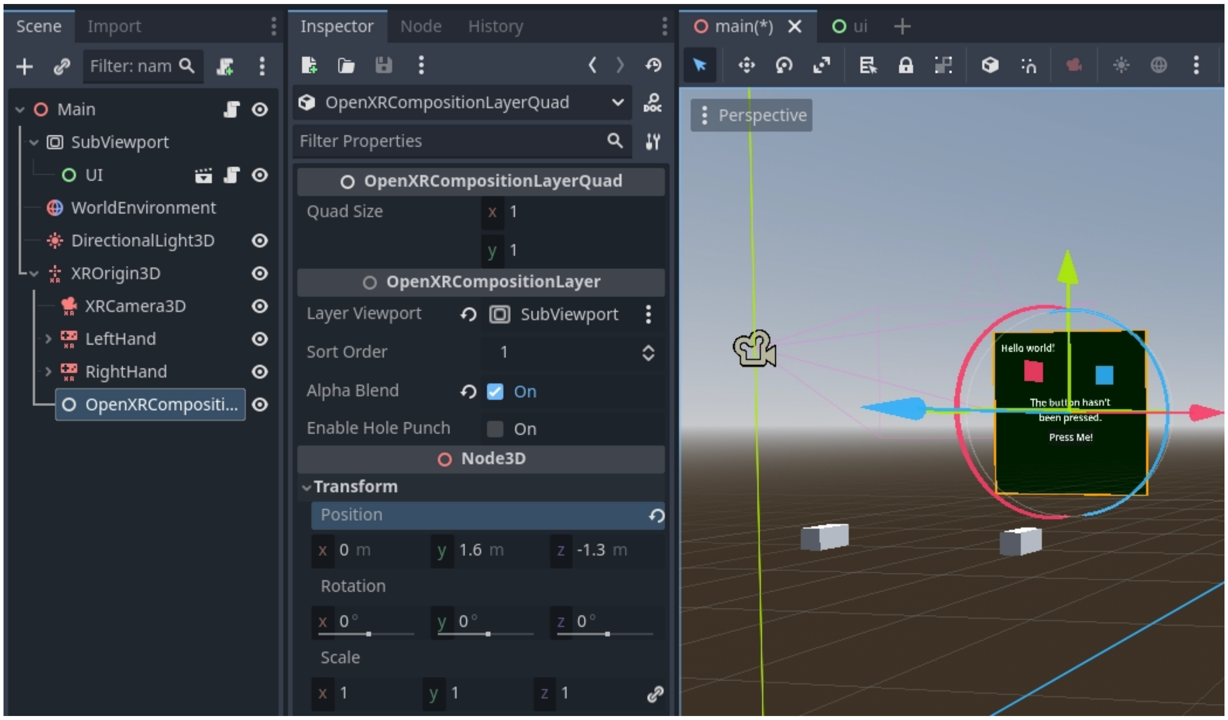
The update mode "When Visible" will not work as Godot can't determine whether the viewport is visible to the user. When assigning our viewport to a composition layer Godot will automatically adjust this.

Adding a composition layer

The second step is adding our composition layer. We simply add the correct composition layer node as a child node of our XROrigin3D node. This is very important as the XR runtime positions everything in relation to our origin.

We want to position the composition layer so it is at eye height and roughly 1 to 1.5 meters away from the player.

We now assign the SubViewport to the `Layer Viewport` property and enable Alpha Blend.



Scene
Import
Filter: nam
Main
SubViewport
UI
WorldEnvironment
DirectionalLight3D
XROrigin3D
XRCamera3D
LeftHand
RightHand
OpenXRCompositi...
Inspector
Node
History
Filter Properties
main(*)
ui
Perspective
Hello world!
The button hasn't been pressed.
Press Me!

OpenXRCompositionLayerQuad

OpenXRCompositionLayerQuad

Quad Size x 1 y 1

OpenXRCompositionLayer

Layer Viewport: SubViewport

Sort Order: 1

Alpha Blend: ☑ On

Enable Hole Punch: ☐ On

Node3D

Transform

Position: x 0 m, y 1.6 m, z -1.3 m

Rotation: x 0°, y 0°, z 0°

Scale: x 1, y 1, z 1

Note

As the player can walk away from the origin point, you will want to reposition the composition layer when the player recenters the view. Using the reference space `Local Floor` will apply this logic automatically.

Making the interface work

So far we're only displaying our UI, to make it work we need to add some code. For this example we're going to keep things simple and make one of the controllers work as a pointer. We'll then simulate mouse actions with this pointer.

This code also requires a `MeshInstance3D` node called `Pointer` to be added as a child to our `OpenXRCompositionLayerQuad` node. We configure a `SphereMesh` with a radius `0.01` meters. We'll be using this as a helper to visualize where the user is pointing.

The main function that drives this functionality is the `intersects_ray` function on our composition layer node. This function takes the global position and orientation of our pointer and returns the UV where our ray intersects our viewport. It returns `Vector2(-1.0, -1.0)` if we're not pointing at our viewport.

We start with setting up some variables, important here are the export variables which identify our controller node with which we point to our screen.

```
extends OpenXRCompositionLayerQuad

const NO_INTERSECTION = Vector2(-1.0, -1.0)

@export var controller : XRController3D
@export var button_action : String = "trigger_click"

var was_pressed : bool = false
var was_intersect : Vector2 = NO_INTERSECTION

...
```

Next we define a helper function that takes the value returned from `intersects_ray` and gives us the global position for that intersection point. This implementation only works for our `OpenXRCompositionLayerQuad` node.

```
...

func _intersect_to_global_pos(intersect : Vector2) -> Vector3:
    if intersect != NO_INTERSECTION:
        var local_pos : Vector2 = (intersect - Vector2(0.5,
0.5)) * quad_size
        return global_transform * Vector3(local_pos.x, -
local_pos.y, 0.0)
    else:
        return Vector3()

...
```

We also define a helper function that takes our `intersect` value and returns our location in the viewports local coordinate system:

```
...

func _intersect_to_viewport_pos(intersect : Vector2) ->
Vector2i:
    if layer_viewport and intersect != NO_INTERSECTION:
        var pos : Vector2 = intersect *
Vector2(layer_viewport.size)
        return Vector2i(pos)
    else:
        return Vector2i(-1, -1)

...
```

The main logic happens in our `_process` function. Here we start by hiding our pointer, we then check if we have a valid controller and viewport, and we call `intersects_ray` with the position and orientation of our controller:

```
...

# Called every frame. 'delta' is the elapsed time since the
previous frame.
func _process(_delta):
    # Hide our pointer, we'll make it visible if we're
interacting with the viewport.
    $Pointer.visible = false

    if controller and layer_viewport:
        var controller_t : Transform3D =
controller.global_transform
        var intersect : Vector2 =
intersects_ray(controller_t.origin, -controller_t.basis.z)

...
```

Next we check if we're intersecting with our viewport. If so, we check if our button is pressed and place our pointer at our intersection point.

```
...

        if intersect != NO_INTERSECTION:
            var is_pressed : bool =
controller.is_button_pressed(button_action)

            # Place our pointer where we're pointing
            var pos : Vector3 =
_intersect_to_global_pos(intersect)
            $Pointer.visible = true
            $Pointer.global_position = pos

...
```

If we were intersecting in our previous process call and our pointer has moved, we prepare a InputEventMouseMotion object to simulate our mouse moving and send that to our viewport for further processing.

```
...

            if was_intersect != NO_INTERSECTION and intersect
!= was_intersect:
                # Pointer moved
                var event : InputEventMouseMotion =
InputEventMouseMotion.new()
                var from : Vector2 =
_intersect_to_viewport_pos(was_intersect)
                var to : Vector2 =
_intersect_to_viewport_pos(intersect)
                if was_pressed:
                    event.button_mask = MOUSE_BUTTON_MASK_LEFT
                event.relative = to - from
                event.position = to
                layer_viewport.push_input(event)

...
```

If we've just released our button we also prepare a InputEventMouseButton object to simulate a button release and send that to our viewport for further processing.

```
...

            if not is_pressed and was_pressed:
                # Button was let go?
                var event : InputEventMouseButton =
InputEventMouseButton.new()
                event.button_index = 1
                event.pressed = false
                event.position =
_intersect_to_viewport_pos(intersect)
                layer_viewport.push_input(event)

...
```

Or if we've just pressed our button we prepare a InputEventMouseButton object to simulate a button press and send that to our viewport for further processing.

```
...

            elif is_pressed and not was_pressed:
                # Button was pressed?
                var event : InputEventMouseButton =
InputEventMouseButton.new()
                event.button_index = 1
                event.button_mask = MOUSE_BUTTON_MASK_LEFT
                event.pressed = true
                event.position =
_intersect_to_viewport_pos(intersect)
                layer_viewport.push_input(event)

...
```

Next we remember our state for next frame.

```
...

            was_pressed = is_pressed
            was_intersect = intersect

...
```

Finally, if we aren't intersecting, we simply clear our state.

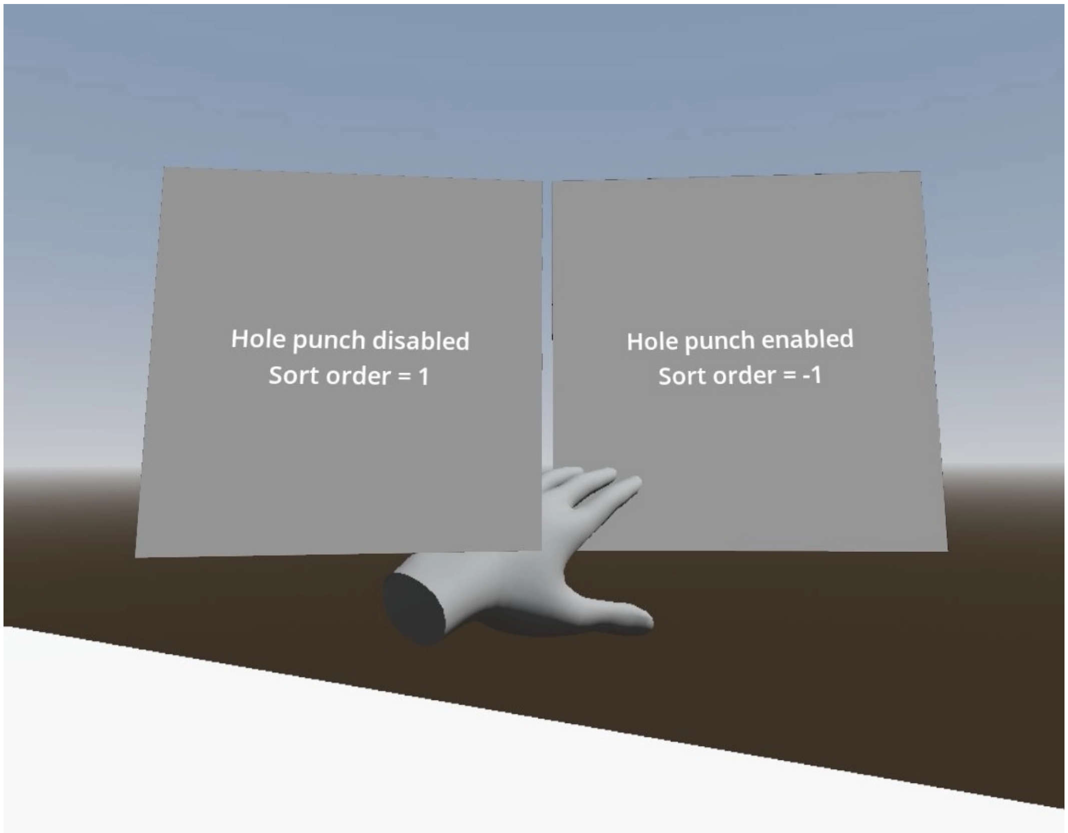
```
...

        else:
            was_pressed = false
            was_intersect = NO_INTERSECTION
```

Hole punching

As the composition layer is composited on top of the render result, it can be rendered in front of objects that are actually forward of the viewport.

By enabling hole punch you instruct Godot to render a transparent object where our viewport is displayed. It does this in a way that fills the depth buffer and clears the current rendering result. Anything behind our viewport will now be cleared, while anything in front of our viewport will be rendered as usual.

You also need to set `Sort Order` to a negative value, the XR compositor will now draw the viewport first, and then overlay our rendering result.


Hole punch disabled
Sort order = 1
Hole punch enabled
Sort order = -1

Use case showing how the users hand is incorrectly obscured by a composition layer when hole punching is not used.

OpenXR hand tracking

Introduction

Note

This page focuses specifically on the feature set exposed through OpenXR. Parts of the functionality presented here also applies to WebXR and can by provided by other XR interfaces.

When discussing hand tracking it is important to know that there are differences of opinion as to where lines are drawn. The practical result of this is that there are differences in implementation between the different OpenXR runtimes. You may find yourself in a place where chosen hardware doesn't support a piece of the puzzle or does things differently enough from the other platforms that you need to do extra work.

That said, recent improvements to the OpenXR specification are closing these gaps and as platforms implement these improvements we are getting closer to a future where we have either full portability between platforms or at least a clear way to detect the capabilities of a platform.

When we look at the early days of VR the focus of the major platforms was on tracked controller based input. Here we are tracking a physical device that also has buttons for further input. From the tracking data we can infer the location of the player's hands but no further information is known, traditionally it was left up to the game to implement a mechanism to display the player's hand and animate the fingers based on further input from the controller, be it due to buttons being pressed or through proximity sensors.

Often fingers are also placed based on context, what the user is holding, and what action a user is performing.

More recently optical hand tracking has become a popular solution, where cameras track the user's hands and full tracking data for the hand and finger positions becomes available. Many vendors saw this as completely separate from controller tracking and introduced independent APIs to access hand and finger positions and orientation data. When handling input, it was up to the game developer to implement a gesture detection mechanism.

This split also exists in OpenXR, where controller tracking is handled primarily by the action map system, while optical hand tracking is primarily handled by the hand tracking API extension.

However, the world is not that black and white and we're seeing a number of scenarios where we cross the line:

- Devices that fit in both categories, such as tracked gloves and controllers such as the Index controller that also perform finger tracking.
- XR Runtimes that implement inferred hand tracking from controller data as a means to solve proper finger placement for multiple controllers.
- XR applications that wish to seamlessly switch between controller and hand tracking offering the same user experience regardless of approach used.

OpenXR is answering this call by introducing further extensions that lets us query the capabilities of the XR runtime/hardware or that add further functionality across this divide. The problem that currently does remain is that there are gaps in adopting these extensions, with some platforms thus not reporting capabilities to their full extent. As such you may need to test for the features available on specific hardware and adjust your approach accordingly.

Demo project

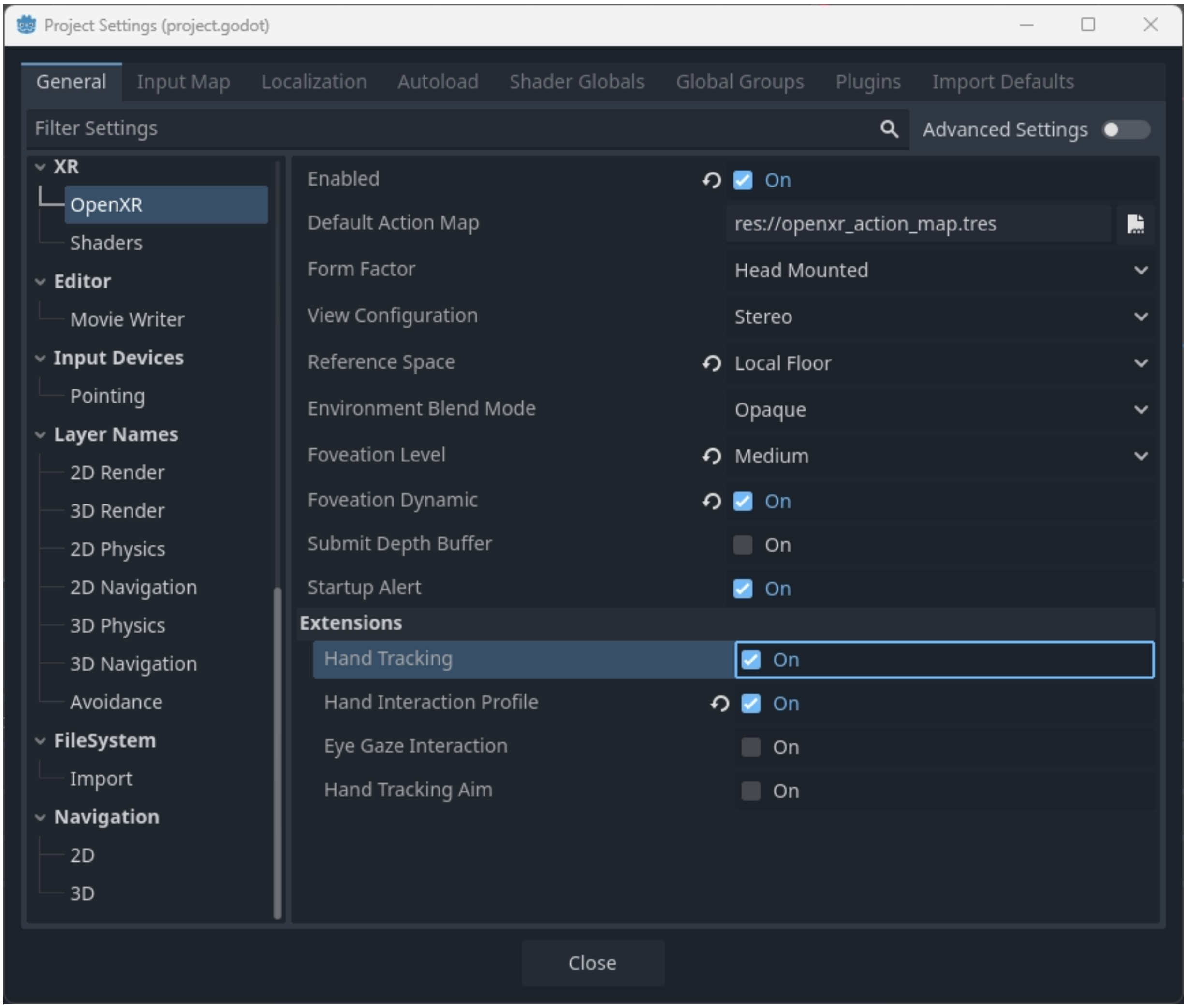
The information presented on this page was used to create a demo project that can be found here [https://github.com/godotengine/godot-demo-projects/tree/master/xr/openxr_hand_tracking_demo].

The Hand Tracking API

As mentioned in our introduction, the hand tracking API is primarily used with optical hand tracking and on many platforms only works when the user is not holding a controller. Some platforms support controller inferred hand tracking meaning that you will get hand tracking data even if the user is holding a controller. This includes SteamVR, Meta Quest (currently native only but Meta link support is likely coming), and hopefully soon others as well.

The hand tracking implementation in Godot has been standardized around the Godot Humanoid Skeleton and works both in OpenXR and WebXR. The instructions below will thus work in both environments.

In order to use the hand tracking API with OpenXR you first need to enable it. This can be done in the project settings:


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Global Groups
Plugins
Import Defaults
Filter Settings
Advanced Settings
XR
OpenXR
Shaders
Editor
Movie Writer
Input Devices
Pointing
Layer Names
2D Render
3D Render
2D Physics
2D Navigation
3D Physics
3D Navigation
Avoidance
FileSystem
Import
Navigation
2D
3D
Close

Enabled ☑ On

Default Action Map res://openxr_action_map.tres

Form Factor Head Mounted

View Configuration Stereo

Reference Space Local Floor

Environment Blend Mode Opaque

Foveation Level Medium

Foveation Dynamic ☑ On

Submit Depth Buffer ☐ On

Startup Alert ☑ On

Extensions

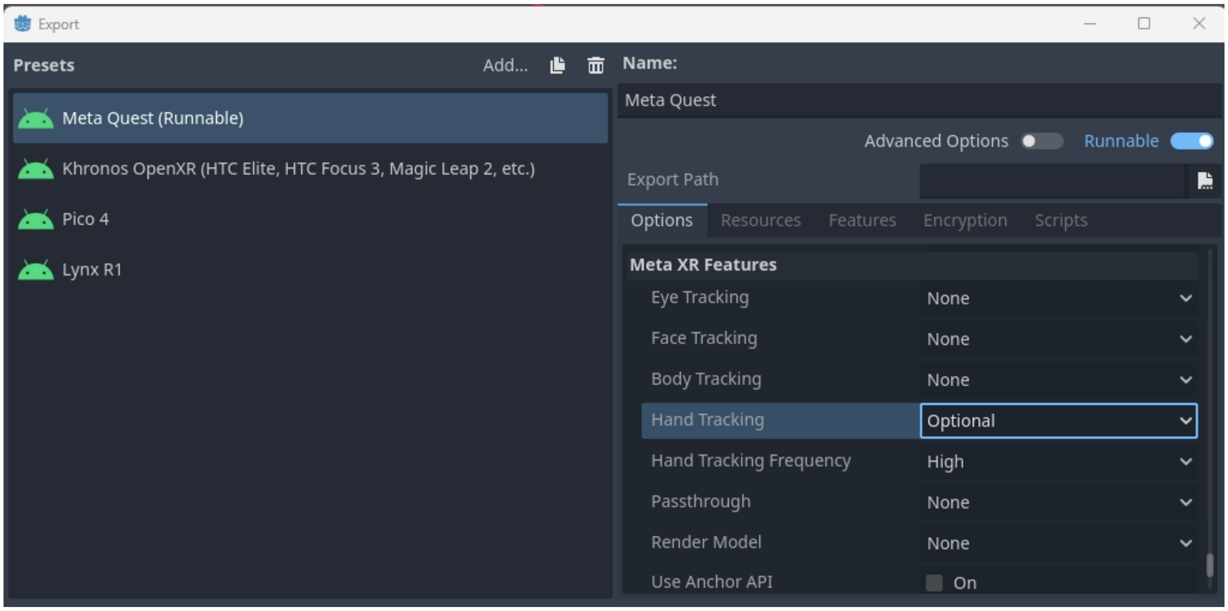
Hand Tracking ☑ On

Hand Interaction Profile ☑ On

Eye Gaze Interaction ☐ On

Hand Tracking Aim ☐ On

For some standalone XR devices you also need to configure the hand tracking extension in export settings, for instance for Meta Quest:


Export
Presets
Add...
Options
Resources
Features
Encryption
Scripts

Meta Quest (Runnable)

Khronos OpenXR (HTC Elite, HTC Focus 3, Magic Leap 2, etc.)

Pico 4

Lynx R1

Name: Meta Quest

Advanced Options ○ Runnable ●

Export Path

Meta XR Features

Eye Tracking: None

Face Tracking: None

Body Tracking: None

Hand Tracking: Optional

Hand Tracking Frequency: High

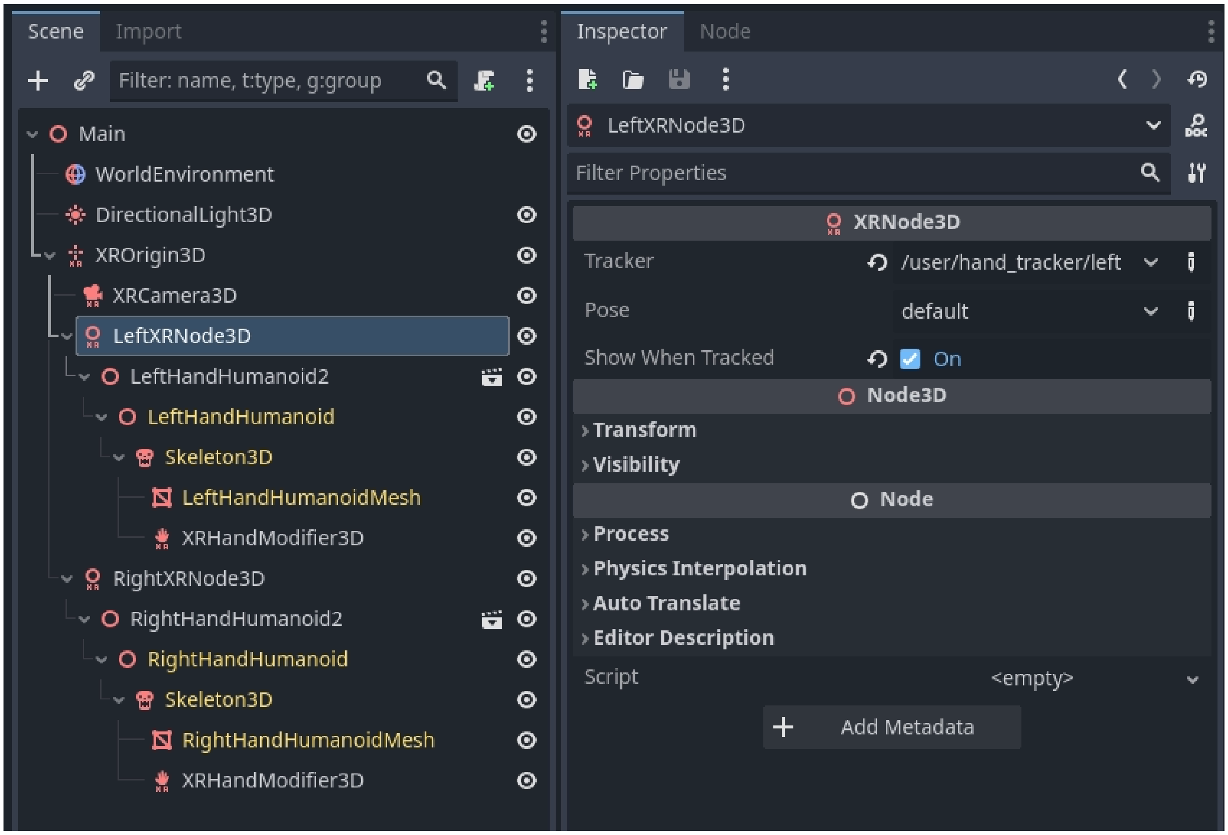
Passthrough: None

Render Model: None

Use Anchor API: ☐ On

Now you need to add 3 components into your scene for each hand:

- A tracked node to position the hand.
- A properly skinned hand mesh with skeleton.
- A skeleton modifier that applies finger tracking data to the skeleton.


Scene
Import
Filter: name, t:type, g:group
Main
WorldEnvironment
DirectionalLight3D
XROrigin3D
XRCamera3D
LeftXRNode3D
LeftHandHumanoid2
LeftHandHumanoid
Skeleton3D
LeftHandHumanoidMesh
XRHandModifier3D
RightXRNode3D
RightHandHumanoid2
RightHandHumanoid
Skeleton3D
RightHandHumanoidMesh
XRHandModifier3D
Inspector
Node
LeftXRNode3D
Filter Properties
XRNode3D
Node3D
Transform
Visibility
Node
Process
Physics Interpolation
Auto Translate
Editor Description
Add Metadata

Tracker /user/hand_tracker/left

Pose default

Show When Tracked ☑ On

Script <empty>

Hand tracking node

The hand tracking system uses separate hand trackers to track the position of the player's hands within our tracking space.

This information has been separated out for the following use cases:

- Tracking happens in the local space of the XROrigin3D node. This node must be a child of the *XROrigin3D* node in order to be correctly placed.
- This node can be used as an IK target when an upper body mesh with arms is used instead of separate hand meshes.
- Actual placement of the hands may be loosely bound to the tracking in scenarios such as avatar creation UIs, fake mirrors, or similar situations resulting in the hand mesh and finger tracking being localized elsewhere.

We'll concentrate on the first use case only.

For this you need to add an XRNode3D node to your `XROrigin3D` node.

- On this node the `tracker` should be set to `/user/hand_tracker/left` or `/user/hand_tracker/right` for the left or right hand respectively.
- The `pose` should remain set to `default`, no other option will work here.
- The checkbox `Show When Tracked` will automatically hide this node if no tracking data is available, or make this node visible if tracking data is available.

Rigged hand mesh

In order to display our hand we need a hand mesh that is properly rigged and skinned. For this Godot uses the hand bone structure as defined for the Godot Humanoid but optionally supporting an extra tip bone for each finger.

The OpenXR hand tracking demo [https://github.com/godotengine/godot-demo-projects/tree/master/xr/openxr_hand_tracking_demo] contains example glTF files of properly rigged hands.

We will be using those here and add them as a child to our `XRNode3D` node. We also need to enable editable children to gain access to our Skeleton3D node.

The hand skeleton modifier

Finally we need to add a XRHandModifier3D node as a child to our `Skeleton3D` node. This node will obtain the finger tracking data from OpenXR and apply it the hand model.

You need to set the `Hand Tracker` property to either `/user/hand_tracker/left` or `/user/hand_tracker/right` depending on whether we are apply the tracking data of respectively the left or right hand.

You can also set the `Bone Update` mode on this node.

- `Full` applies the hand tracking data fully. This does mean that the skeleton positioning will potentially reflect the size of the actual hand of the user. This can lead to scrunching effect if meshes aren't weighted properly to account for this. Make sure you test your game with players of all sizes when optical hand tracking is used!
- `Rotation Only` will only apply rotation to the bones of the hands and keep the bone length as is. In this mode the size of the hand mesh doesn't change.

With this added, when we run the project we should see the hand correctly displayed if hand tracking is supported.

The hand tracking data source

This is an OpenXR extension that provides information about the source of the hand tracking data. At this moment only a few runtimes implement it but if it is available, Godot will activate it.

If this extension is not supported and thus unknown is returned, you can make the following assumptions:

- If you are using SteamVR (including Steam link), only controller based hand tracking is supported.
- For any other runtime, if hand tracking is supported, only optical hand tracking is supported (Note, Meta Link currently fall into this category).
- In all other cases, no hand tracking is supported at all.

You can access this information through code:

```
var hand_tracker : XRHandTracker =
XRServer.get_tracker('/user/hand_tracker/left')
if hand_tracker:
    if hand_tracker.has_tracking_data:
        if hand_tracker.hand_tracking_source ==
XRHandTracker.HAND_TRACKING_SOURCE_UNKNOWN:
            print("Hand tracking source unknown")
        elif hand_tracker.hand_tracking_source ==
XRHandTracker.HAND_TRACKING_SOURCE_UNOBSTRUCTED:
            print("Hand tracking source is optical hand
tracking")
        elif hand_tracker.hand_tracking_source ==
XRHandTracker.HAND_TRACKING_SOURCE_CONTROLLER:
            print("Hand tracking data is inferred from
controller data")
        else:
            print("Unknown hand tracking source ",
hand_tracker.hand_tracking_source)
    else:
        print("Hand is currently not being tracked")
else:
    print("No hand tracker registered")
```

This example simply logs the state for the left hand.

If in this example no hand tracker is returned by `get_tracker`, this means the hand tracking API is not supported on the XR runtime at all.

If there is a tracker but *has_tracking_data* is false, the user's hand is currently not being tracked. This is likely caused by one of the following reasons:

- The player's hand is not visible by any of the tracking cameras on the headset
- The player is currently using a controller and the headset only supports optical hand tracking

- The controller is turned off and only controller hand tracking is supported.

Handling user input

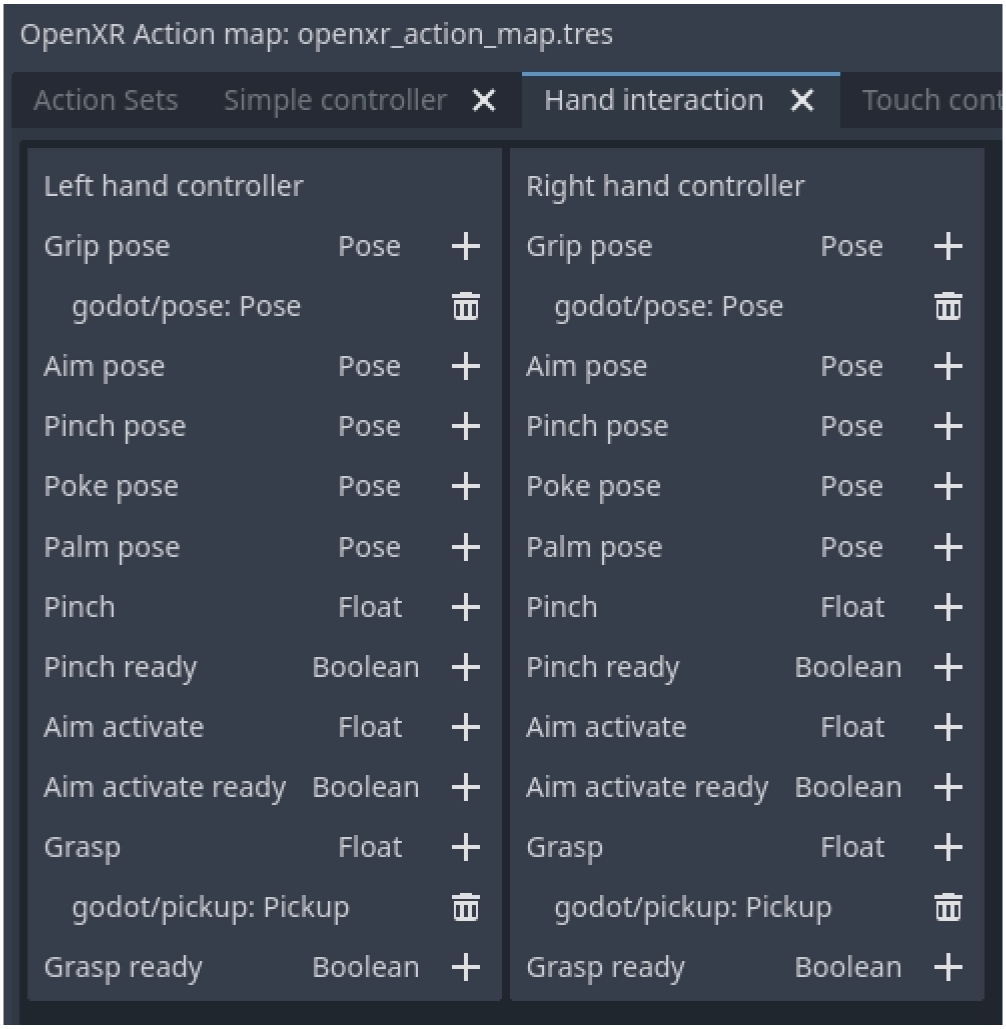
Reacting to actions performed by the user is handled through The XR action map if controllers are used. In the action map you can map various inputs like the trigger or joystick on the controller to an action. This can then drive logic in your game.

When hand tracking is used we originally had no such inputs, inputs are driven by gestures made by the user such as making a fist to grab or pinching the thumb and index finger together to select something. It was up to the game developer to implement this.

Recognizing that there is an increasing demand for applications that can switch seamlessly between controller and hand tracking and the need some form of basic input capability, a number of extensions were added to the specification that provide some basic gesture recognition and can be used with the action map.

The hand interaction profile

The hand interaction profile extension [https://github.khronos.org/OpenXR-Inventory/extension_support.html#XR_EXT_hand_interaction] is a new core extension which supports pinch, grasp, and poke gestures and related poses. There is still limited support for this extension but it should become available in more runtimes in the near future.



OpenXR Action map: openxr_action_map.tres

Left hand controller		Right hand controller	
Grip pose	Pose	Grip pose	Pose
godot/pose: Pose		godot/pose: Pose	
Aim pose	Pose	Aim pose	Pose
Pinch pose	Pose	Pinch pose	Pose
Poke pose	Pose	Poke pose	Pose
Palm pose	Pose	Palm pose	Pose
Pinch	Float	Pinch	Float
Pinch ready	Boolean	Pinch ready	Boolean
Aim activate	Float	Aim activate	Float
Aim activate ready	Boolean	Aim activate ready	Boolean
Grasp	Float	Grasp	Float
godot/pickup: Pickup		godot/pickup: Pickup	
Grasp ready	Boolean	Grasp ready	Boolean

The pinch gesture is triggered by pinching your thumb and index finger together. This is often used as a select gesture for menu systems, similar to using your controller to point at an object and press the trigger to select and is thus often mapped as such.

- The `pinch pose` is a pose positioned in the middle between the tip of the thumb and the tip of the index finger and oriented such that a ray cast can be used to identify a target.
- The `pinch` float input is a value between 0.0 (the tip of the thumb and index finger are apart) and 1.0 (the tip of the thumb and index finger are touching).

- The `pinch ready` input is true when the tips of the fingers are (close to) touching.

The grasp gesture is triggered by making a fist and is often used to pick items up, similar to engaging the squeeze input on controllers.

- The `grasp` float input is a value between 0.0 (open hand) and 1.0 (fist).
- The `grasp ready` input is true when the user made a fist.

The poke gesture is triggered by extending your index finger, this one is a bit of an exception as the pose at the tip of your index finger is often used to poke an interactable object. The `poke pose` is a pose positioned on the tip of the index finger.

Finally the `aim activate (ready)` input is defined as an input that is 1.0/true when the index finger is extended and pointing at a target that can be activated. How runtimes interpret this, is not clear.

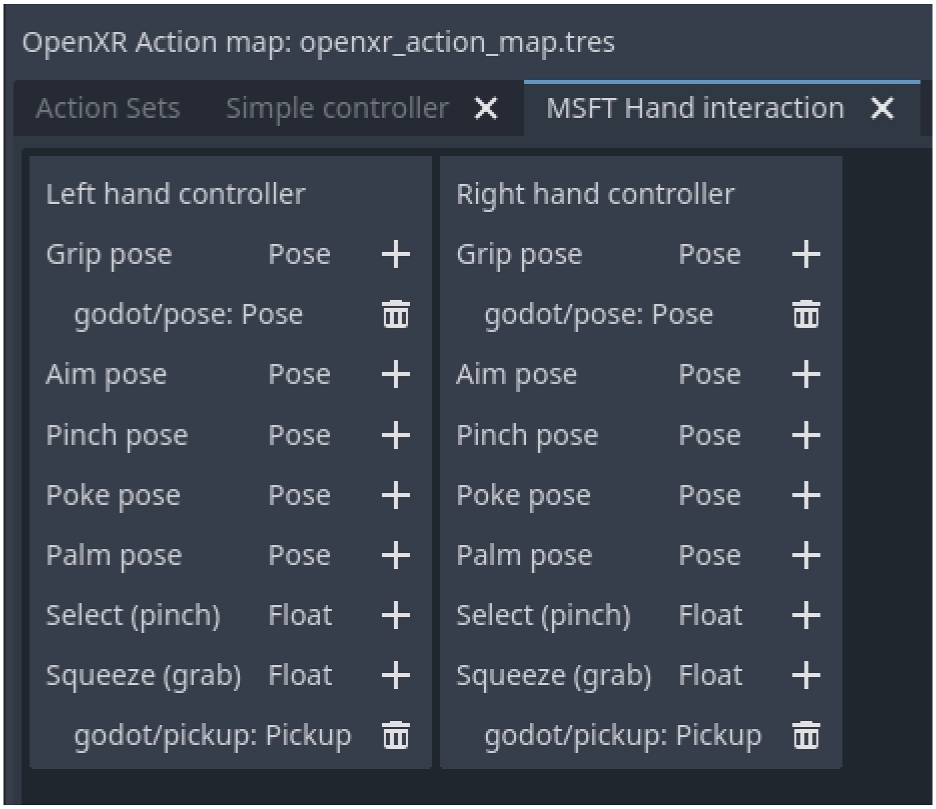
With this setup the normal `left_hand` and `right_hand` trackers are used and you can thus seamlessly switch between controller and hand tracking input.

Note

You need to enable the hand interaction profile extension in the OpenXR project settings.

Microsoft hand interaction profile

The Microsoft hand interaction profile extension [https://github.khronos.org/OpenXR-Inventory/extension_support.html#XR_MSFT_hand_interaction] was introduced by Microsoft and loosely mimics the simple controller profile. Meta has also added support for this extension but only on their native OpenXR client, it is currently not available over Meta Link.



Left hand controller		Right hand controller	
Grip pose	Pose	Grip pose	Pose
godot/pose: Pose		godot/pose: Pose	
Aim pose	Pose	Aim pose	Pose
Pinch pose	Pose	Pinch pose	Pose
Poke pose	Pose	Poke pose	Pose
Palm pose	Pose	Palm pose	Pose
Select (pinch)	Float	Select (pinch)	Float
Squeeze (grab)	Float	Squeeze (grab)	Float
godot/pickup: Pickup		godot/pickup: Pickup	

Pinch support is exposed through the `select` input, the value of which is 0.0 when the tip of the thumb and index finger are apart and 1.0 when they are together.

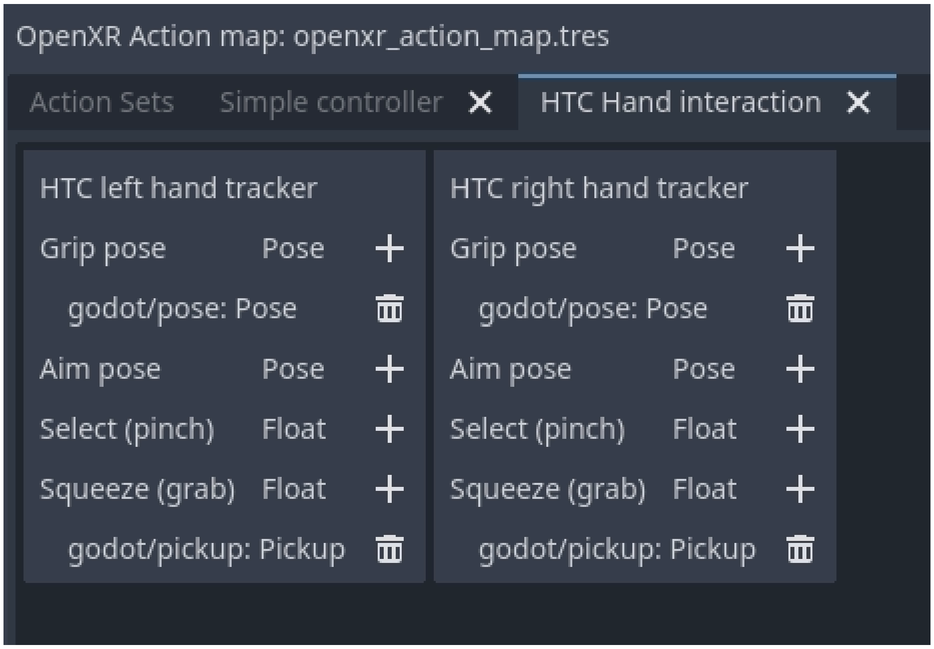
Note that in this profile the `aim pose` is redefined as a pose between thumb and index finger, oriented so a ray cast can be used to identify a target.

Grasp support is exposed through the `squeeze` input, the value of which is 0.0 when the hand is open, and 1.0 when a fist is made.

With this setup the normal `left_hand` and `right_hand` trackers are used and you can thus seamlessly switch between controller and hand tracking input.

HTC hand interaction profile

The [HTC hand interaction profile extension](https://github.khronos.org/OpenXR-Inventory/extension_support.html#XR_HTC_hand_interaction) [https://github.khronos.org/OpenXR-Inventory/extension_support.html#XR_HTC_hand_interaction] was introduced by HTC and is defined similarly to the Microsoft extension. It is only supported by HTC for the Focus 3 and Elite XR headsets.



HTC left hand tracker
Grip pose Pose
godot/pose: Pose
Aim pose Pose
Select (pinch) Float
Squeeze (grab) Float
godot/pickup: Pickup

HTC right hand tracker
Grip pose Pose
godot/pose: Pose
Aim pose Pose
Select (pinch) Float
Squeeze (grab) Float
godot/pickup: Pickup

See the Microsoft hand interaction profile for the gesture support.

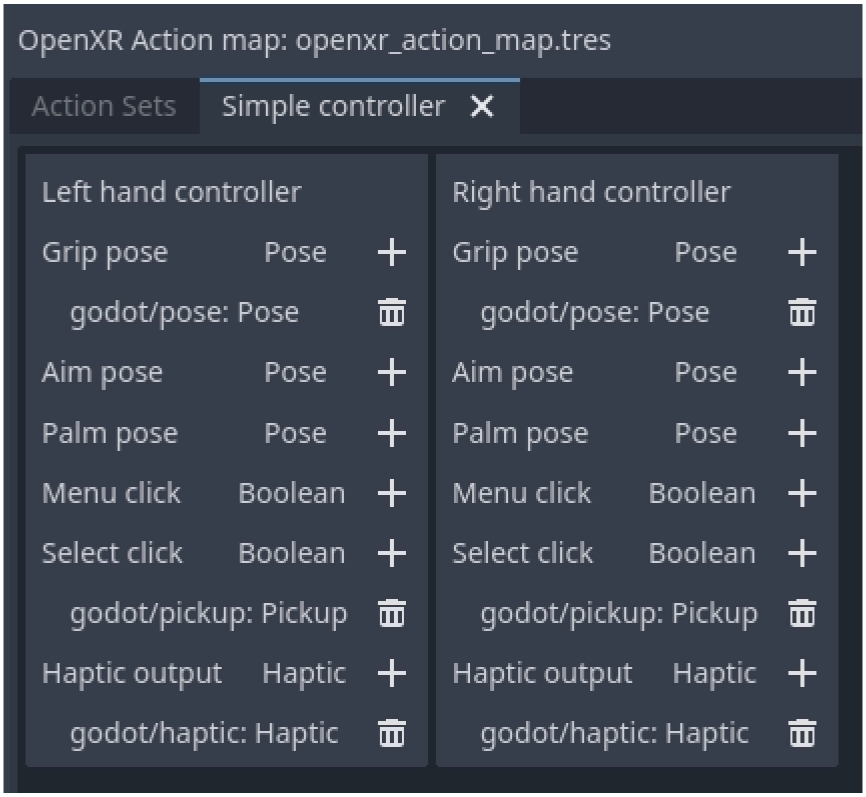
The defining difference is that this extension introduces two new trackers, `/user/hand_htc/left` and `/user/hand_htc/right`. This means that extra logic needs to be implemented to switch between the default trackers and the HTC specific trackers when the user puts down, or picks up, their controller.

Simple controller profile

The simple controller profile is a standard core profile defined as a fallback profile when a controller is used for which no profile exists.

There are a number of OpenXR runtimes that will mimic controllers through the simple controller profile when hand tracking is used.

Unfortunately there is no sound way to determine whether an unknown controller is used or whether hand tracking is emulating a controller through this profile.



OpenXR Action map: openxr_action_map.tres

Left hand controller		Right hand controller	
Grip pose	Pose	Grip pose	Pose
godot/pose: Pose		godot/pose: Pose	
Aim pose	Pose	Aim pose	Pose
Palm pose	Pose	Palm pose	Pose
Menu click	Boolean	Menu click	Boolean
Select click	Boolean	Select click	Boolean
godot/pickup: Pickup		godot/pickup: Pickup	
Haptic output	Haptic	Haptic output	Haptic
godot/haptic: Haptic		godot/haptic: Haptic	

XR runtimes are free to define how the simple controller profile operates, so there is also no certainty to how this profile is mapped to gestures.

The most common mapping seems to be that `select click` is true when the tip of the thumb and index fingers are touching while the user's palm is facing away from the user. `menu click` will be true when tip of the thumb and index fingers are touching while the user's palm is facing towards the user.

With this setup the normal `left_hand` and `right_hand` trackers are used and you can thus seamlessly switch between controller and hand tracking input.

Note

As some of these interaction profiles have overlap it is important to know that you can add each profile to your action map and the XR runtime will choose the best fitting profile.

For instance, a Meta Quest supports both the Microsoft hand interaction profile and simple controller profile. If both are specified the Microsoft hand interaction profile will take precedence and will be used.

The expectation is that once Meta supports the core hand interaction profile extension, that profile will take precedence over both Microsoft and simple controller profiles.

Gesture based input

If the platform doesn't support any interaction profiles when hand tracking is used, or if you're building an application where you need more complicated gesture support you're going to need to build your own gesture recognition system.

You can obtain the full hand tracking data through the XRHandTracker resource for each hand. You can obtain the hand tracker by calling `XRServer.get_tracker` and using either `/user/hand_tracker/left` or `/user/hand_tracker/left` as the tracker. This resource provides access to all the joint information for the given hand.

Detailing out a full gesture recognition algorithm goes beyond the scope of this manual however there are a number of community projects you can look at:

- Julian Todd's Auto hands library [https://github.com/Godot-Dojo/Godot-XR-AH]
- Malcolm Nixons Hand Pose Detector [https://github.com/Malcolmnixon/GodotXRHandPoseDetector]

OpenXR body tracking

Support for full body tracking in OpenXR is only just becoming available for a select few platforms. As support solidifies information will be added to this page.

HTC Tracker support

An option that has been available for some time is doing full body tracking using HTC trackers. These are currently supported through SteamVR and on HTC Elite XR headsets. They are exposed through the action map system.

These trackers are identified by their roles which are assigned to them when configured. Simply add XRController3D nodes as children to the XROrigin3D node and assign one of the following trackers:

HTC trackers
/user/vive_tracker_htcx/role/handheld_object
/user/vive_tracker_htcx/role/left_foot
/user/vive_tracker_htcx/role/right_foot
/user/vive_tracker_htcx/role/left_shoulder
/user/vive_tracker_htcx/role/right_shoulder

/user/vive_tracker_htcx/role/left_elbow
/user/vive_tracker_htcx/role/right_elbow
/user/vive_tracker_htcx/role/left_knee
/user/vive_tracker_htcx/role/right_knee
/user/vive_tracker_htcx/role/waist
/user/vive_tracker_htcx/role/chest
/user/vive_tracker_htcx/role/camera
/user/vive_tracker_htcx/role/keyboard

You can now use these as targets for IK modifiers on a full body avatar.

Introducing XR tools

Out of the box Godot gives you all the basic support to setup an XR project. XR specific game mechanics however need to be implemented on top of this foundation. While Godot makes this relatively easy this can still be a daunting task.

For this reason Godot has developed a toolkit called Godot XR Tools [https://github.com/GodotVR/godot-xr-tools] that implements many of the basic mechanics found in XR games, from locomotion to object interaction to UI interaction.

This toolkit is designed to work with both OpenXR and WebXR runtimes. We'll be using this as a base for our documentation here. It helps developers hit the ground running but for more specific use cases building your own logic is just as valid. In that case XR tools can help in providing inspiration.

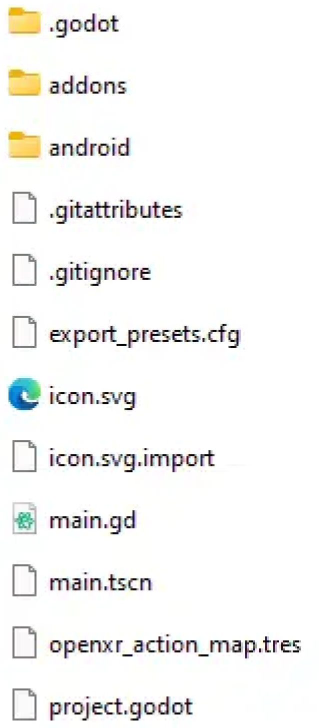
Note

The current stable release of Godot XR Tools available on the Asset Library is the version for Godot 3. The port of Godot XR Tools for Godot 4 is available for download from the github website. We will walk through the step of downloading and installing it here. We will update the documentation once Godot XR Tools becomes available for Godot 4 through the asset library.

Installing XR Tools

Continuing on from our project we started in Setting up XR we want to add in the Godot XR Tools library. This can be downloaded from the Godot XR Tools releases page [https://github.com/GodotVR/godot-xr-tools/releases]. Simply find the latest pre-release for Godot 4 and then under assets download the `godot-xr-tools.zip` file.

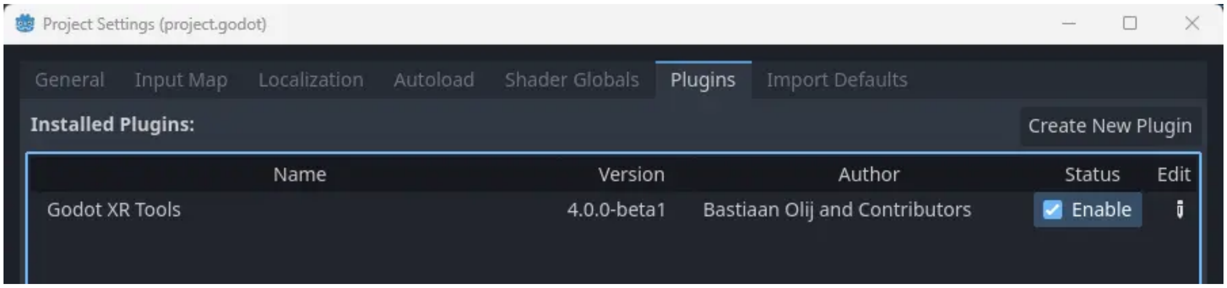
Once downloaded unzip the file. You will notice the files are held within a `godot-xr-tools` subfolder. Inside of this folder you will find an `addons` folder. It is this folder that you want to copy in its entirety to your Godot project folder, your project should now look something like this:



.godot
addons
android
.gitattributes
.gitignore
export_presets.cfg
icon.svg
icon.svg.import
main.gd
main.tscn
openxr_action_map.tres
project.godot

Now open up your project in Godot, if you haven't already, and give it a minute or so to import all the resources of the plugin.

Next open the `Project` menu and select `Project Settings...` Now go to the `Plugins` tab and enable the plugin.


Project Settings (project.godot)
General
Input Map
Localization
Autoload
Shader Globals
Plugins
Import Defaults
Installed Plugins:
Create New Plugin

Name	Version	Author	Status	Edit
Godot XR Tools	4.0.0-beta1	Bastiaan Olij and Contributors	☑ Enable	

Basic hands

Just to get a feel of things we're going to add a few standard components that dress up our scene starting with hands for our player.

OpenXR supports full hand tracking however there currently are significant differences in capabilities between the different XR Runtimes.

As a reliable alternative Godot XR Tools comes with a number of rigged hand scenes that react on trigger and grip inputs of your controller. These hands come in low and high poly versions, come in a few configurations, a number of animation files to control finger positions and a number of different textures.

In your scene tree select your left hand XRController3D node. Now click on the `instantiate Child Scene` button to add a child scene. And select `addons/godot-xr-tools/hands/scenes/lowpoly/left_hand_low.tscn`.

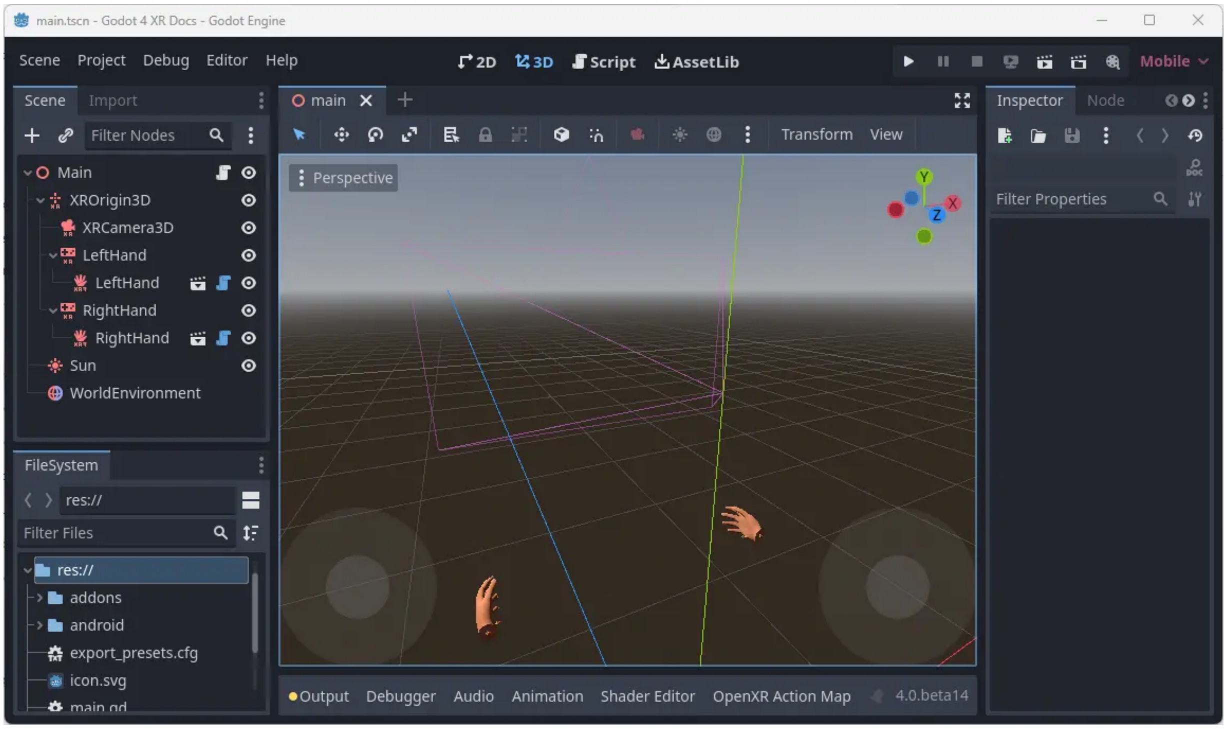
As you can see from the path of this scene, low poly models are in the `lowpoly` subfolder while high poly models are in the `highpoly` subfolder. You will want to use the low poly versions if you plan to release your game on mobile devices.

The default hand we chose is just a hand. The other options are:

- tac_glove - the hand is wearing a glove with fingers exposed
- full_glove - the hand is wearing a glove that covers the entire hand

Finally each hand comes in a `physics` version. This exposes all the bones. We'll look at how that can be used in another tutorial.

We repeat the same for the right hand.


main.tscn - Godot 4 XR Docs - Godot Engine
Scene Project Debug Editor Help
2D 3D Script AssetLib
Mobile
Scene Import
Filter Nodes
Main
XROrigin3D
XRCamera3D
LeftHand
LeftHand
RightHand
RightHand
Sun
WorldEnvironment
main
Transform View
Perspective
Inspector Node
Filter Properties
FileSystem
res://
Filter Files
res://
addons
android
export_presets.cfg
icon.svg
main.gd
Output Debugger Audio Animation Shader Editor OpenXR Action Map
4.0.beta14

More information

We'll continue with adding features to our tutorial project using Godot XR tools in the next couple of pages. More detailed information about the toolkit can be found on the toolkits help pages [https://godotvr.github.io/godot-xr-tools/].

Basic XR Locomotion

For basic locomotion we're going to continue using our Godot XR Tools library. The library contains both basic movement features as more advanced features.

Adding our player body

The first step we need to do is to add a helper node to our XROrigin3D node. Because XR supports roomscale tracking you can't simply add your XR setup to a CharacterBody3D node and expect things to work. You will run into trouble when the user moves around their physical space and is no longer standing in the center of their room. Godot XR Tools embeds the needed logic into a helper node called `PlayerBody`.

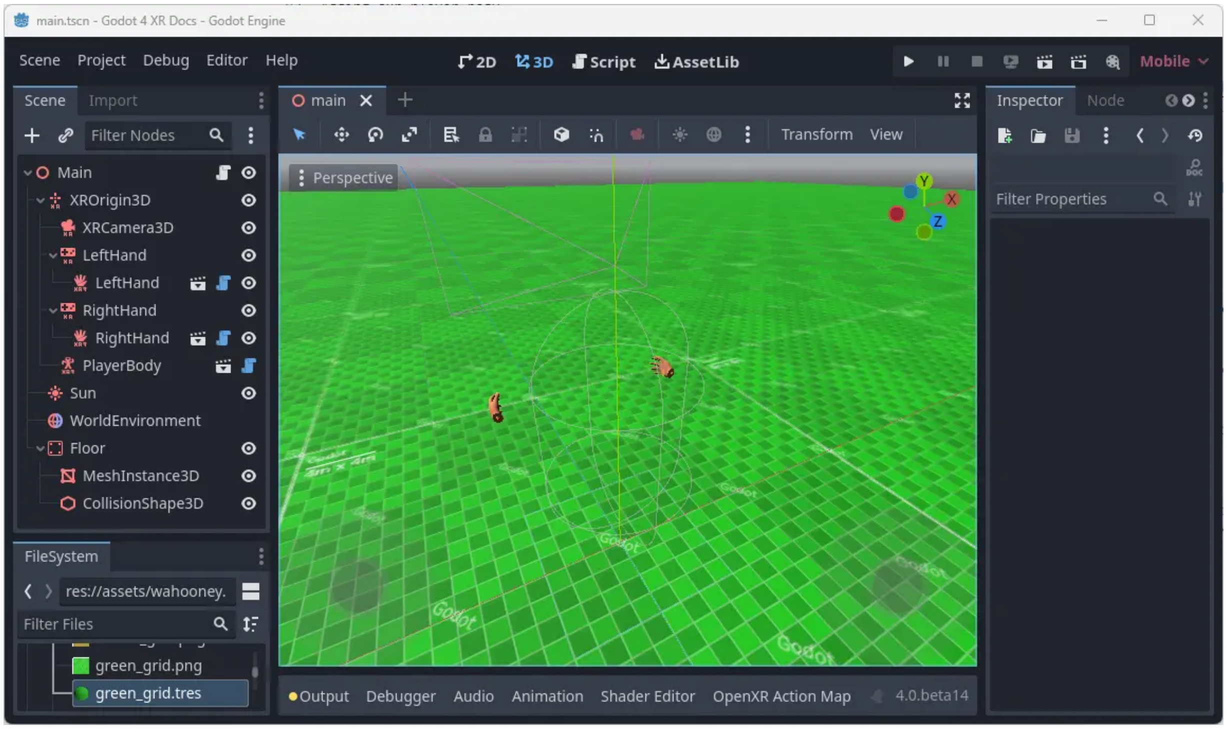
Select your XROrigin3D node and click on the `instantiate Child Scene` button to add a child scene. Select `addons/godot-xr-tools/player/player_body.tscn` and add this node.

Adding a floor

This node governs the in game movement of your character and will immediately react to gravity. So to prevent our player from infinitely falling down we'll quickly add a floor to our scene.

We start by adding a StaticBody3D node to our root node and we rename this to `Floor`. We add a MeshInstance3D node as a child node for our `Floor`. Then create a new PlaneMesh as it's mesh. For now we set the size of the mesh to 100 x 100 meters. Next we add a CollisionShape3D node as a child node for our `Floor`. Then create a `BoxShape` as our shape. We set the size of this box shape to 100 x 1 x 100 meters. We also need to move our collision shape down by 0.5 meters so the top of our box is flush with the floor.

To make it easier to see that we're actually moving around our world, a white floor isn't going to do it. For this we add a material to our PlaneMesh and set the albedo to a grid texture we've created with [Wahooneys excellent free texture generator](https://wahooney.itch.io/texture-grid-generator) [https://wahooney.itch.io/texture-grid-generator].


main.tscn - Godot 4 XR Docs - Godot Engine
Scene
Project
Debug
Editor
Help
2D
3D
Script
AssetLib
Mobile
Scene
Import
Filter Nodes
Main
XROrigin3D
XRCamera3D
LeftHand
LeftHand
RightHand
RightHand
PlayerBody
Sun
WorldEnvironment
Floor
MeshInstance3D
CollisionShape3D
FileSystem
res://assets/wahooney.
Filter Files
green_grid.png
green_grid.tres
main
Transform
View
Perspective
Output
Debugger
Audio
Animation
Shader Editor
OpenXR Action Map
4.0.beta14
Inspector
Node
Filter Properties

Direct movement

We're going to start adding some basic direct movement to our setup. This allows the user to move through the virtual world using joystick input.

Note

It is important to note that moving through the virtual world while the player is standing still in the real world, can be nausea inducing especially for players who are new to VR. The default settings on our movement functions are fairly conservative. We advise you to stick to these defaults but offer features in game to enable less comfortable settings for more experienced users who are used to playing VR games.

We want to enable this on the right hand controller. We do this by adding a subscene to the right hand XRController3D node. Select `addons/godot-xr-tools/functions/movement_direct.tscn` as the scene to add.

This function adds forward and backwards movement to the player by using the joystick on the right hand controller. It has an option to also add left/right strafe but by default this is disabled.

Instead, we are going to add the ability for the player to also turn with this joystick. We will add another subscene to our controller node, select `addons/godot-xr-tools/functions/movement_turn.tscn` for this.

The turn system by default uses a snap turn approach. This means that turning happens in steps. This may seem jarring however it is a tried and tested method of combating motion sickness. You can easily switch to a mode that offers smooth turning by changing the `mode` property on the turn node.

If you run your game at this point in time you will find that you can move through the world freely using the right hand joystick.

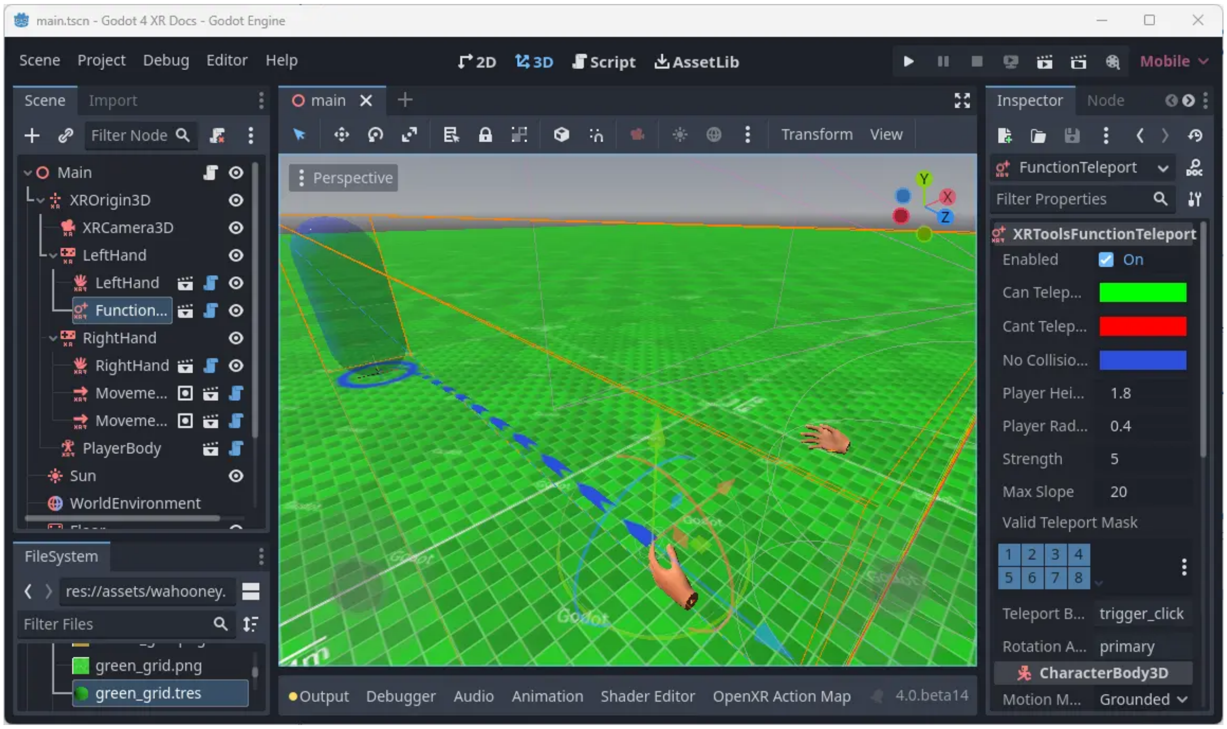
Teleport

An alternative to direct movement that some users find more pleasant is the ability to teleport to another location within your game world. Godot XR Tools supports this through the teleport function and we will be adding this to our left hand controller.

Add a new child scene to your left hand [XRController3D]() node by selecting the `addons/godot-xr-tools/functions/function_teleport.tscn` scene.

With this scene added the player will be able to teleport around the world by pressing the trigger on the left hand controller, pointing where they want to go, and then releasing the trigger. The player can also adjust the orientation by using the left hand controllers joystick.

If you've followed all instructions correctly your scene should now look something like this:


main.tscn - Godot 4 XR Docs - Godot Engine
Scene Project Debug Editor Help
2D 3D Script AssetLib
Mobile
Scene Import
Filter Node
Main
XROrigin3D
XRCamera3D
LeftHand
LeftHand
Function...
RightHand
RightHand
Moveme...
Moveme...
PlayerBody
Sun
WorldEnvironment
FileSystem
res://assets/wahooney.
Filter Files
green_grid.png
green_grid.tres
main
Transform View
Perspective
Output Debugger Audio Animation Shader Editor OpenXR Action Map
4.0.beta14
Inspector Node
FunctionTeleport
Filter Properties

XRToolsFunctionTeleport
Enabled ☑ On
Can Telep... 🟩
Cant Telep... 🟥
No Collisio... 🟦
Player Hei... 1.8
Player Rad... 0.4
Strength 5
Max Slope 20
Valid Teleport Mask 1 2 3 4 5 6 7 8
Teleport B... trigger_click
Rotation A... primary
CharacterBody3D
Motion M... Grounded

More advanced movement features

Godot XR Tools adds many more movement features such as gliding, a grapple hook implementation, a jetpack, climbing mechanics, etc.

Most work similarly to the basic movement features we've handled so far, simply add the relevant subscene from the plugin to the controller that implements it.

We'll look at some of these in more detail later on in this tutorial where additional setup is required (such as climbing) but for others please look at Godot XR Tools own help pages for details.

How to contribute

The Godot Engine is free and open-source. Like any community-driven project, we rely on volunteer contributions. On this page we want to showcase the various ways you as users can participate - to help you find the right starting place with the skillset you have. Because contrary to popular opinion, we need more than just programmers on the project!

Fundraising

- **Donate**

  We created the non-profit Godot Foundation [https://godot.foundation/] to be able to support the Godot Engine in both matters of finance and administration. In practice, this means the Foundation hires people to work part-time or full-time on the project. These jobs include engine development as well as related tasks like code reviews, production management, community & marketing efforts, and more.

  With as little as 5 EUR per month, you can help us keep going strong. Currently, we are intending to hire more core developers, as to cover more ground with full-time specialists that supplement and guide volunteer work.

  Join the Development Fund [https://fund.godotengine.org]

- **Donation Drives** Think about your followers on social media, or other communities you are active in. Use that reach to remind your social environment that even small contributions can make a difference, especially when done by a great number of people at the same time.

Are you a content creator? Consider adding a link to the Godot Development Fund [https://fund.godotengine.org] to your descriptions. If you do live streams, perhaps think about organizing a stream with donation incentives.

- **Publish Godot Games.** You heard right, simply publishing a game #MadeWithGodot can positively impact the well-being of this project. Your personal success elevates the engine to a viable alternative for other developers, growing the community further. Additionally, it opens the doors for us to approach industry contacts about possible cooperations.

Technical contributions

- **Report bugs & other issues** As active users of the engine, you are better equipped to identify bugs and other issues than anyone else. To let us know about your findings, fill out this bug report form [https://github.com/godotengine/godot/issues/new/choose] on our GitHub. Make sure to include as much information as possible to ensure these issues can easily be reproduced by others.

  If you are interested in helping keep our bug tracker organized, you can even join the bugsquad [https://chat.godotengine.org/channel/bugsquad]!

- **Test Development Versions** While it is recommended to use the stable releases for your projects, you can help us test dev releases, betas, and release candidates by opening a copy of your project in them and checking what problems this introduces or maybe even solves. Make sure to have a backup ready, since this can produce irreversible changes.

  Find recent development versions [https://godotengine.org/download/preview/] directly on our download page, or linked in their own blog posts.

- **Contribute Engine Code (mainly C++)** The engine development is mainly coordinated on our Contributor RocketChat [https://chat.godotengine.org/], so if you are serious about making PRs you should join us there!

  Read more about the **technical submission process**: Contributing code

  For each subject area of the engine, there is a corresponding team to coordinate the work. Join the linked chat to get more eyes on your related PR, learn about open todos, or partake in meetings. For some areas, specialists might even be encouraged to step up as maintainer! List of teams [https://godotengine.org/teams/]

- **Review Code Contributions** All pull requests need to be thoroughly reviewed before they can be merged into the master branch. Help us get a headstart by participating in the code review process.

  To get started, chose any open pull request [https://github.com/godotengine/godot/pulls] and reference our **style guide**: Pull request review process

- **Write Plugins (GDScript, C#, & more)** Community addons are not directly included in the core engine download or repository, yet they provide essential quality of life upgrades for your fellow game developers. Upload your plugins to the Godot Asset Library [https://godotengine.org/asset-library/asset] to make them available to others.

- **Demo projects (GDScript, C#, and making Assets)** We provide new users with demo projects [https://github.com/godotengine/godot-demo-projects/] so they can quickly test new features or get familiar with the engine in the first place. At industry events, we might even exhibit these demo projects to showcase what Godot can do! Help improve existing projects or supply your own to be added to the pool, and join the demo channel [https://chat.godotengine.org/channel/demo-content] in the Contributor RocketChat to talk about it.

- **Documentation** The documentation is one of the most essential parts of any tech project, yet the need to document new features and substantial changes often gets overlooked. Join the documentation team [https://chat.godotengine.org/channel/documentation] to improve the Godot Engine with your technical writing skills.
- **Translations (spoken languages other than English)** Are you interested in making the Godot Engine more accessible to non-English speakers? Contribute to our community-translations [https://hosted.weblate.org/projects/godot-engine/godot/].

Community support

- **Call for Moderators** With a community of our size, we need people to step up as volunteer moderators in all kinds of places. These teams are organized by the Godot Foundation, but would not function without the dedication of active community members like you.

  Have a look around your favorite community platform and you might come across open application calls.
- **Answer tech-support questions** With many new people discovering the Godot Engine recently, the need for peer-to-peer tech-support has never been greater. See the Godot website [https://godotengine.org/community] for a list of official and user-supported Godot communities.

- **Create tutorials & more** How did you get started with the Godot Engine? Chances are you looked for learning materials outside of what the documentation provides. Without content creators covering the game development process, there would not be this big of a community today. Therefore it seemed only right to mention them in a page about important contributions to the project.

Contribution workflow

Godot is a large project, both in terms of the codebase and the workload for contributors. The guides below serve as a helper when you get stuck dealing with Git or GitHub, or if you are generally interested in how maintainers approach the project.

- Contributing code
- Testing and reporting issues
- Contributing to the documentation
- Contributing translations
- Bisecting regressions
- Bug triage guidelines
- Pull request workflow
- Pull request review process
- Testing pull requests

Contributing code

The possibility to study, use, modify and redistribute modifications of the engine's source code are the fundamental rights that Godot's MIT [https://tldrlegal.com/license/mit-license] license grants you, making it free and open source software [https://en.wikipedia.org/wiki/Free_and_open-source_software].

As such, everyone is entitled to modify Godot's source code [https://github.com/godotengine/godot], and send those modifications back to the upstream project in the form of a patch (a text file describing the changes in a ready-to-apply manner) or - in the modern workflow that we use - via a so-called "pull request" (PR), i.e. a proposal to directly merge one or more Git commits (patches) into the main development branch.

Contributing code changes upstream has two big advantages:

- Your own code will be reviewed and improved by other developers, and will be further maintained directly in the upstream project, so you won't have to reapply your own changes every time you move to a newer version. On the other hand it comes with a responsibility, as your changes have to be generic enough to be beneficial to all users, and not just your project; so in some cases it might still be relevant to keep your changes only for your own project, if they are too specific.
- The whole community will benefit from your work, and other contributors will behave the same way, contributing code that will be beneficial to you. At the time of this writing, over 2,000 developers have contributed code changes to the engine!

To ensure good collaboration and overall quality, the Godot developers enforce some rules for code contributions, for example regarding the style to use in the C++ code (indentation, brackets, etc.) or the Git and PR workflow.

A good place to start is by searching for issues tagged as good first issue [https://github.com/godotengine/godot/issues?q=is%3Aissue+is%3Aopen+label%3A%22good+first+issue%22] on GitHub.

See also

Technical details about the PR workflow are outlined in a specific section, Pull request workflow.

Details about the code style guidelines and the `clang-format` tool used to enforce them are outlined in Code style guidelines.

All pull requests must go through a review process before being accepted. Depending on the scope of the changes, it may take some time for a maintainer responsible for the modified part of the engine to provide their review. We value all of our contributors and ask them to be patient in the meantime, as it is expected that in an open source project like Godot, there is going to be way more contributions than people validating them.

To make sure that your time and efforts aren't wasted, it is recommended to vet the idea first before implementing it and putting it for a review as a PR. To that end, Godot has a proposal system [https://github.com/godotengine/godot-proposals]. Its usage is encouraged to plan changes and discuss them with the community. Implementation details can also be discussed with other contributors on the Godot Contributors Chat [https://chat.godotengine.org/].

Note

Proposals are only required when working on an enhancement or a new feature. Bug reports are sufficient for fixing issues.

Testing and reporting issues

Another great way of contributing to the engine is to test development releases or the development branch and to report issues. It is also helpful to report issues discovered in stable releases, so that they can be fixed in the development branch and in future maintenance releases.

Testing development versions

To help with the testing, you have several possibilities:

- Compile the engine from source yourself, following the instructions of the Compiling page for your platform.
- Test official pre-release binaries when they are announced (usually on the blog and other community platforms), such as alpha, beta and release candidate (RC) builds.
- Test "trusted" unofficial builds of the development branch; just ask community members for reliable providers. Whenever possible, it's best to use official binaries or to compile yourself though, to be sure about the provenance of your binaries.

As mentioned previously, it is also helpful to keep your eyes peeled for potential bugs that might still be present in the stable releases, especially when using some niche features of the engine which might get less testing by the developers.

Filing an issue on GitHub

Godot uses GitHub's issue tracker [https://github.com/godotengine/godot/issues] for bug reports. When you start filing a bug report, you’ll be given a form to fill out. Please try to follow it so that all issues are consistent and provide the required information.

Contributing to the documentation

There are two separate resources referred to as "documentation" in Godot:

- **The class reference.** This is the documentation for the complete Godot API as exposed to GDScript and the other scripting languages. It can be consulted offline, directly in Godot's code editor, or online at Godot Class Reference. To contribute to the class reference, you have to edit the XML file corresponding to the class and make a pull request. See Contributing to the class reference and Class reference primer for more details.
- **The tutorials and engine documentation and its translations.** This is the part you are reading now, which is distributed in the HTML format. Its contents are generated from plain text files in the reStructured Text (rst) format, to which you can contribute via pull requests on the godot-docs [https://github.com/godotengine/godot-docs] GitHub repository. See Contributing to the documentation for more details.

Contributing translations

To make Godot accessible to everyone, including users who may prefer resources in their native language instead of English, our community helps translate both the Godot editor and its documentation in many languages.

See Editor and documentation localization for more details.

Bisecting regressions

Bisecting is a way to find regressions in software. After reporting a bug on the Godot repository on GitHub [https://github.com/godotengine/godot], you may be asked by a contributor to *bisect* the issue. Bisecting makes it possible for contributors to fix bugs faster, as they can know in advance which commit caused the regression. Your effort will be widely appreciated :)

The guide below explains how to find a regression by bisecting.

What is bisecting?

Godot developers use the Git [https://git-scm.com/] version control system. In the context of Git, bisecting is the process of performing a manual binary search [https://en.wikipedia.org/wiki/Binary_search_algorithm] to determine when a regression appeared. While it's typically used for bugs, it can also be used to find other kinds of unexpected changes such as performance regressions.

Using official builds to speed up bisecting

Before using Git's `bisect` command, we strongly recommend trying to reproduce the bug with an older (or newer) official release. This greatly reduces the range of commits that potentially need to be built from source and tested. You can find binaries of official releases, as well as alphas, betas, and release candidates here [https://downloads.tuxfamily.org/godotengine/].

If you have experience with Godot 3.x and can reproduce an issue with Godot 4.0, we recommend trying to reproduce the issue in the latest Godot 3.x version (if the feature exhibiting the bug is present in 3.x). This can be used to check whether the issue is a regression in 4.0 or not.

- If the issue **is present** in 3.x, then you'll need to check whether the issue occurs in older 3.x versions as well.
- If the issue is **not present** in 3.x, then you can try older 4.0 alphas and betas to determine when the regression started.

Warning

Project files may be incompatible between Godot versions. **Make a backup of your project** before starting the bisection process.

Going from the oldest to the newest build generally reduces the risk of the project not being able to successfully open in the editor, thanks to backwards compatibility. Try to reduce your project to the smallest repeatable example too. The more minimal the project is, the more likely you'll be able to open it without compatibility issues in newer engine versions.

The Git bisect command

If you've found a build that didn't exhibit the bug in the above testing process, you can now start bisecting the regression. The Git version control system offers a built-in command for this: `git bisect`. This makes the process semi-automated as you only have to build the engine, run it and try to reproduce the bug.

Note

Before bisecting a regression, you need to set up a build environment to compile Godot from source. To do so, read the Compiling page for your target platform. (Compiling Godot from source doesn't require C++ programming knowledge.)

Note that compiling Godot can take a while on slow hardware (up an hour for each full rebuild on a slow dual-core CPU). This means the full process can take up to several hours. If your hardware is too slow, you may want to stop there and report the results of your "pre-bisecting" on the GitHub issue so another contributor can continue bisecting from there.

Determine the commit hashes

To start bisecting, you must first determine the commit hashes (identifiers) of the "bad" and "good" build. "bad" refers to the build that exhibits the bug, whereas "good" refers to the version that doesn't exhibit the bug. If you're using a pre-release build as the "good" or "bad" build, browse the download mirror [https://downloads.tuxfamily.org/godotengine/], go to the folder that contains the pre-release you downloaded and look for the `README.txt` file. The commit hash is written inside that file.

If you're using a stable release as the "good" or "bad" build, use one of the following commit hashes depending on the version:

```
4.1.1-stable
4.1-stable
4.0.3-stable
4.0.2-stable
4.0.1-stable
4.0-stable
3.5.2-stable
3.5.1-stable
3.5-stable
3.4.5-stable
```

```
3.4.4-stable
3.4.3-stable
3.4.2-stable
3.4.1-stable
3.4-stable
3.3.4-stable
3.3.3-stable
3.3.2-stable
3.3.1-stable
3.3-stable
3.2-stable
3.1-stable
3.0-stable
```

You can also use this Bash function to retrieve the Git commit hash of a pre-release build (add it to your `$HOME/.bashrc` or similar):

```
gd_snapshot_commit() {
    curl -s
https://downloads.tuxfamily.org/godotengine/$1/$2/README.txt \
        | grep 'from commit' \
        | sed 's/^Built from commit \(.*\)\.$/\1/'
}
```

Example usage:

```
gd_snapshot_commit 4.0 beta4
```

To refer to the latest state of the master branch, you can use `master` instead of a commit hash. Note that unlike tagged releases or snapshot commit hashes, `master` is a perpetually moving target.

Build the engine

Get Godot's source code using Git. Once this is done, in the terminal window, use `cd` to reach the Godot repository folder and enter the following command:

```
# <good commit hash> is hash of the build that works as
expected.
# <bad commit hash> is hash of the build exhibiting the bug.
git bisect start
git bisect good <good commit hash>
git bisect bad <bad commit hash>
```

Compile Godot. This assumes you've set up a build environment:

```
scons
```

Run the engine

Run the binary located in the `bin/` folder and try to reproduce the bug.

Note

Double-check the output file name in `bin/` to make sure you're actually running the binary you've just compiled. Different Godot versions will output binaries with different names.

If the build **still** exhibits the bug, run the following command:

```
git bisect bad
```

If the build **does not** exhibit the bug, run the following command:

```
git bisect good
```

After entering one of the commands above, Git will switch to a different commit. You should now build Godot again, try to reproduce the bug, then enter `git bisect good` or `git bisect bad` depending on the result. You'll have to repeat this several times. The longer the commit range, the more steps will be required. 5 to 10 steps are usually sufficient to find most regressions; Git will remind you of the number of steps remaining (in the worst case scenario).

Once you've completed enough steps, Git will display the commit hash where the regression appeared. Write this commit hash as a comment to the GitHub issue you've bisected. This will help in solving the issue. Thanks again for contributing to Godot :)

Note

You can read the full documentation on `git bisect` here [https://git-scm.com/docs/git-bisect].

Bug triage guidelines

This page describes the typical workflow of the bug triage team aka bugsquad when handling issues and pull requests on Godot's GitHub repository [https://github.com/godotengine/godot]. It is bound to evolve together with the bugsquad, so do not hesitate to propose modifications to the following guidelines.

Issues management

GitHub proposes various features to manage issues:

- Set one or several labels from a predefined list
- Set one milestone from a predefined list
- Keep track of the issue in the project dashboard
- Define one contributor as "assignee" among the Godot engine organization members

As the Godot engine organization on GitHub currently has a restricted number of contributors, we do not use assignees extensively for now. All contributors are welcome to take on any issue, if relevant after mentioning it on the issue ticket and/or discussing the best way to resolve it with other developers.

For the time being, we do not use the project dashboard feature either.

As far as possible, we try to assign labels (and milestones, when relevant) to both issues and pull requests.

Labels

The following labels are currently defined in the Godot repository:

Categories:

- *Archived*: either a duplicate of another issue, or invalid. Such an issue would also be closed.
- *Breaks compat*: describes something that can only be fixed by breaking compatibility with existing projects.
- *Bug*: describes something that is not working properly.
- *Cherrypick*: describes something that can be backported to a stable branch after being merged in the `master` branch.
- *Confirmed*: has been confirmed by at least one other contributor than the bug reporter (typically for *Bug* reports). The purpose of this label is to let developers know which issues are still reproducible when they want to select what to work on. It is therefore a good practice to add in a comment on what platform and what version or commit of Godot the issue could be reproduced; if a developer looks at the issue one year later, the *Confirmed* label may not be relevant anymore.
- *Crash:* describes a bug that causes the engine to crash. This label is only used for "hard" crashes, not freezes.
- *Discussion*: the issue is not consensual and needs further discussion to define what exactly should be done to address the topic.
- *Documentation*: issue related to the documentation. Mainly to request enhancements in the API documentation. Issues related to the ReadTheDocs documentation should be filed on the godot-docs [https://github.com/godotengine/godot-docs] repository.
- *Enhancement*: describes a proposed enhancement to an existing functionality.
- *Feature proposal*: describes a wish for a new feature to be implemented. Note that the main Godot repository no longer accepts feature requests. Please use godot-proposals [https://github.com/godotengine/godot-proposals] instead.
- *For PR meeting*: the issue needs to be discussed in a pull request meeting. These meetings are public and are held on the Godot Contributors Chat [https://chat.godotengine.org/].

- *Good first issue*: the issue is *assumed* to be an easy one to fix, which makes it a great fit for new contributors who want to become familiar with the code base. It should be removed while an active PR is available, that resolves this issue.
- *High priority:* the issue is particularly important as it can prevent people from releasing their projects or cause data loss.
- *Needs testing*: the issue/pull request could not be completely tested and thus need further testing. This can mean that it needs to be tested on different hardware/software configurations or even that the steps to reproduce are not certain.
- *Needs work*: the pull request needs additional work before it can be merged.
- *Performance*: issues that directly impact engine or editor performance. Can also be used for pull requests that improve performance or add low-end-friendly options. Should not be coupled with *Usability*.
- *Production*: Relates to the production team.
- *Regression*: the bug appeared after a stable release not exhibiting the bug was released.
- *Salvageable*: the pull request can't be merged due to design issues or merge conflicts and its author is not active anymore. However, it can still be picked up by an external contributor to bring it to a mergeable state. To do so, you need to open a new pull request based on the original pull request.
- *Tracker*: issue used to track other issues (like all issues related to the plugin system).
- *Usability*: issues that directly impact user usability. Should not be coupled with *Performance*.

The categories are used for general triage of the issues. They can be combined in some way when relevant, e.g. an issue can be labelled *Enhancement* and *Usability* at the same time if it's an issue to improve usability. Or *Feature proposal* and *Discussion* if it's a non-consensual feature request, or one that is not precise enough to be worked on. At least one of the categories *Bug*, *Enhancement* or *Discussion* is used to describe an issue or pull request.

Topics:

- *2D*: relates to 2D-specific issues. Should be coupled with one of the labels below, and should not be coupled with *3D*.
- *3D*: relates to 3D-specific issues. Should be coupled with one of the labels below, and should not be coupled with *2D*.
- *Animation*: relates to the Animation system, editors and importers.
- *Assetlib*: relates to issues with the asset library.
- *Audio*: relates to the audio features (low and high level).
- *Buildsystem*: relates to building issues, either linked to the SCons buildsystem or to compiler peculiarities.
- *Codestyle*: relates to the programming style used within the codebase.
- *Core*: anything related to the core engine. Specific topics are split off separately as they crop up.
- *Dotnet*: relates to the C# / Dotnet bindings.
- *Editor*: relates to issues in the editor (mainly UI).
- *Export*: relates to the export system and templates.
- *GDExtension*: relates to the GDExtension system for native extensions.
- *GDScript*: relates to GDScript.
- *GUI*: relates to GUI (Control) nodes or to Nodes that compose user interfaces.
- *Import*: relates to the resource import system.
- *Input*: relates to the input system.
- *Multiplayer*: relates to multiplayer (high-level networking) systems.
- *Navigation*: relates to the navigation system (including A* and navmeshes).
- *Network*: relates to (lot-level) networking.
- *Particles*: particles, particle systems and their editors.
- *Physics*: relates to the physics engine (2D/3D).

- *Plugin*: relates to problems encountered while writing plugins.
- *Porting*: relates to some specific platforms or exporting projects.
- *Rendering*: relates to the 2D and 3D rendering engines.
- *Shaders*: relates to the Godot shader language or visual shaders.
- *Tests*: relates to unit tests.
- *Thirdparty*: relates to third-party libraries used in Godot.
- *XR*: relates to Augmented Reality or Virtual Reality.

Issues would typically correspond to only one topic, though it's not unthinkable to see issues that fit two bills. The general idea is that there will be specialized contributors teams behind all topics, so they can focus on the issues labelled with their team's topic.

Platforms:

Android, *iOS*, *Linux*, *macOS*, *Web*, *Windows*

By default, it is assumed that a given issue applies to all platforms. If one of the platform labels is used, it is then exclusive and the previous assumption doesn't stand anymore (so if it's a bug on e.g. Android and Linux exclusively, select those two platforms).

Documentation labels

In the documentation repository [https://github.com/godotengine/godot-docs], we use the following labels:

- *Archived*: either a duplicate of another issue, or invalid. Such an issue would also be closed.
- *Bug*: Incorrect information in an existing page. Not to be used for *missing* information.
- *Cherrypick*: describes something that can be backported to a stable branch after being merged in the `master` branch.
- *Dependencies*: describes pull requests that update a dependency file.

- *Discussion*: the issue is not consensual and needs further discussion to define what exactly should be done to address the topic.
- *Enhancement*: new information to be added in an existing page.
- *Good first issue*: the issue is *assumed* to be an easy one to fix, which makes it a great fit for new contributors who want to become familiar with the code base. It should be removed while an active PR is available, that resolves this issue.
- *Needs work*: the pull request needs additional work before it can be merged.
- *Python*: Pull requests that update Python code.
- *Salvageable*: the pull request can't be merged due to design issues or merge conflicts and its author is not active anymore. However, it can still be picked up by an external contributor to bring it to a mergeable state. To do so, you need to open a new pull request based on the original pull request.
- *Tracker*: issue used to track other issues (like all issues related to the plugin system).

Area:

- *About*: Issues and PRs related to the About section of the documentation and other general articles.
- *Class reference*: the issue is about the class reference, not a documentation page.
- *Community*: Issues and PRs related to the Community section of the documentation.
- *Contributing*: Issues and PRs related to the Contributing/Development section of the documentation.
- *Getting started*: Issues and PRs related to the Getting Started section of the documentation.
- *Manual*: Issues and PRs related to the Manual/Tutorials section of the documentation.

Content:

- *Images*: Issues and PRs involving outdated or incorrect images in articles.
- *New page*: Issues and PRs related to creation of new documentation pages for new or undocumented features.
- *Organization*: Issues and PRs related to reorganizing the content.
- *Proofreading*: Issues and PRs related to proofreading the documentation.
- *Redirect*: Issues and PRs involving moving content and adding a redirect rule on the backend.
- *Website*: Issues related to adding website features and fixing bugs, whether on the front or back-end,

Topic:

The available topics describe the same content as the topics in the main repository.

Milestones

Milestones [https://github.com/godotengine/godot/milestones] correspond to planned future versions of Godot for which there is an existing roadmap. Issues that fit in the said roadmap should be filed under the corresponding milestone; if they don't correspond to any current roadmap, they should be left without milestone. As a rule of thumb, an issue corresponds to a given milestone if it concerns a feature that is new in the milestone, or a critical bug that can't be accepted in any future stable release, or anything that Juan wants to work on right now. :)

Contributors are free to pick issues regardless of their assigned milestone; if a fix is proposed for a bug that was not deemed urgent and thus without milestone, it would likely still be very welcome.

Pull request workflow

The so-called "PR workflow" used by Godot is common to many projects using Git, and should be familiar to veteran free software contributors. The idea is that only a small number (if any) commit directly to the *master* branch. Instead, contributors *fork* the project (i.e. create a copy of it, which they can modify as they wish), and then use the GitHub interface to request a *pull* from one of their fork's branches to one branch of the original (often named *upstream*) repository.

The resulting *pull request* (PR) can then be reviewed by other contributors, which might approve it, reject it, or most often request that modifications be done. Once approved, the PR can then be merged by one of the core developers, and its commit(s) will become part of the target branch (usually the *master* branch).

We will go together through an example to show the typical workflow and associated Git commands. But first, let's have a quick look at the organization of Godot's Git repository.

Git source repository

The repository on GitHub [https://github.com/godotengine/godot] is a Git [https://git-scm.com] code repository together with an embedded issue tracker and PR system.

Note

If you are contributing to the documentation, its repository can be found here [https://github.com/godotengine/godot-docs].

The Git version control system is the tool used to keep track of successive edits to the source code - to contribute efficiently to Godot, learning the basics of the Git command line is *highly* recommended. There exist some graphical interfaces for Git, but they usually encourage users to take bad habits regarding the Git and PR workflow, and we therefore recommend not to use them. In particular, we advise not to use GitHub's online editor for code contributions (although it's tolerated for small fixes or documentation changes) as it enforces one commit per file and per modification, which quickly leads to PRs with an unreadable Git history (especially after peer review).

See also

The first sections of Git's "Book" are a good introduction to the tool's philosophy and the various commands you need to master in your daily workflow. You can read them online on the Git SCM [https://git-scm.com/book/en/v2] website. You can also try out GitHub's interactive guide [https://try.github.io/].

The branches on the Git repository are organized as follows:

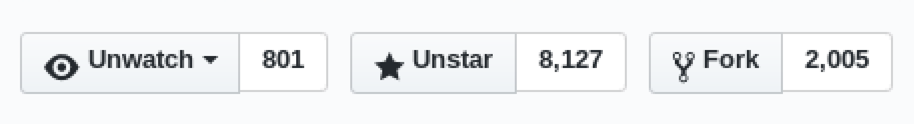
- The `master` branch is where the development of the next major version occurs. As a development branch, it can be unstable and is not meant for use in production. This is where PRs should be done in priority.
- The stable branches are named after their version, e.g. `3.1` and `2.1`. They are used to backport bugfixes and enhancements from the `master` branch to the currently maintained stable release (e.g. 3.1.2 or 2.1.6). As a rule of thumb, the last stable branch is maintained until the next minor version (e.g. the `3.0` branch was maintained until the release of Godot 3.1). If you want to make PRs against a maintained stable branch, please check first if your changes are also relevant for the `master`

branch, and if so make the PR for the `master` branch in priority. Release managers can then cherry-pick the fix to a stable branch if relevant.

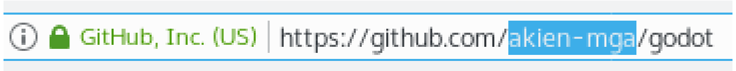
- There might occasionally be feature branches, usually meant to be merged into the `master` branch at some time.

Forking and cloning

The first step is to *fork* the godotengine/godot [https://github.com/godotengine/godot] repository on GitHub. To do so, you will need to have a GitHub account and to be logged in. In the top right corner of the repository's GitHub page, you should see the "Fork" button as shown below:



Click it, and after a while you should be redirected to your own fork of the Godot repo, with your GitHub username as namespace:



You can then *clone* your fork, i.e. create a local copy of the online repository (in Git speak, the *origin remote*). If you haven't already, download Git from its website [https://git-scm.com] if you're using Windows or macOS, or install it through your package manager if you're using Linux.

Note

If you are on Windows, open Git Bash to type commands. macOS and Linux users can use their respective terminals.

To clone your fork from GitHub, use the following command:

```
git clone https://github.com/USERNAME/godot
```

Note

In our examples, the "$" character denotes the command line prompt on typical UNIX shells. It is not part of the command and should not be typed.

After a little while, you should have a `godot` directory in your current working directory. Move into it using the `cd` command:

```
cd godot
```

We will start by setting up a reference to the original repository that we forked:

```
git remote add upstream https://github.com/godotengine/godot
git fetch upstream
```

This will create a reference named `upstream` pointing to the original `godotengine/godot` repository. This will be useful when you want to pull new commits from its `master` branch to update your fork. You have another remote reference named `origin`, which points to your fork (`USERNAME/godot`).

You only need to do the above steps once, as long as you keep that local `godot` folder (which you can move around if you want, the relevant metadata is hidden in its `.git` subfolder).

Note

Branch it, pull it, code it, stage it, commit, push it, rebase it... technologic.

This bad take on Daft Punk's *Technologic* shows the general conception Git beginners have of its workflow: lots of strange commands to learn by copy and paste, hoping they will work as expected. And that's actually not a bad way to learn, as long as you're curious and don't hesitate to question your search engine when lost, so we will give you the basic commands to know when working in Git.

In the following, we will assume as an example that you want to implement a feature in Godot's Project Manager, which is coded in the `editor/project_manager.cpp` file.

Branching

By default, the `git clone` should have put you on the `master` branch of your fork (`origin`). To start your own feature development, we will create a feature branch:

```
# Create the branch based on the current branch (master)
git branch better-project-manager

# Change the current branch to the new one
git checkout better-project-manager
```

This command is equivalent:

```
# Change the current branch to a new named one, based on the
current branch
git checkout -b better-project-manager
```

If you want to go back to the `master` branch, you'd use:

```
git checkout master
```

You can see which branch you are currently on with the `git branch` command:

```
git branch
  2.1
* better-project-manager
  master
```

Be sure to always go back to the `master` branch before creating a new branch, as your current branch will be used as the base for the new one. Alternatively, you can specify a custom base branch after the new branch's name:

```
git checkout -b my-new-feature master
```

Updating your branch

This would not be needed the first time (just after you forked the upstream repository). However, the next time you want to work on something, you will notice that your fork's `master` is several commits behind the upstream `master` branch: pull requests from other contributors would have been merged in the meantime.

To ensure there won't be conflicts between the feature you develop and the current upstream `master` branch, you will have to update your branch by *pulling* the upstream branch.

```
git pull --rebase upstream master
```

The `--rebase` argument will ensure that any local changes that you committed will be re-applied *on top* of the pulled branch, which is usually what we want in our PR workflow. This way, when you open a pull request, your own commits will be the only difference with the upstream `master` branch.

While rebasing, conflicts may arise if your commits modified code that has been changed in the upstream branch in the meantime. If that happens, Git will stop at the conflicting commit and will ask you to resolve the conflicts. You can do so with any text editor, then stage the changes (more on that later), and proceed with `git rebase --continue`. Repeat the operation if later commits have conflicts too, until the rebase operation completes.

If you're unsure about what is going on during a rebase and you panic (no worry, we all do the first few times), you can abort the rebase with `git rebase --abort`. You will then be back to the original state of your branch before calling `git pull --rebase`.

Note

If you omit the `--rebase` argument, you will instead create a merge commit which tells Git what to make of the two distinct branches. If any conflicts arise, they would be resolved all at once via this merge commit.

While this is a valid workflow and the default behavior of `git pull`, merge commits within PRs are frowned upon in our PR workflow. We only use them when merging PRs into the upstream branch.

The philosophy is that a PR should represent the final stage of the changes made to the codebase, and we are not interested in mistakes and fixes that would have been done in intermediate stages before merging. Git gives us great tools to "rewrite the history" and make it as if we got things right the first time, and we're happy to use it to ensure that changes are easy to review and understand long after they have been merged.

If you have already created a merge commit without using `rebase`, or have made any other changes that have resulted in undesired history, the best option is to use an *interactive rebase* on the upstream branch. See the dedicated section for instructions.

Tip

If at any time you want to *reset* a local branch to a given commit or branch, you can do so with `git reset --hard <commit ID>` or `git reset --hard <remote>/<branch>` (e.g. `git reset --hard upstream/master`).

Be warned that this will remove any changes that you might have committed in this branch. If you ever lose commits by mistake, use the `git reflog` command to find the commit ID of the previous state that you would like to restore, and use it as argument of `git reset --hard` to go back to that state.

Making changes

You would then do your changes to our example's `editor/project_manager.cpp` file with your usual development environment (text editor, IDE, etc.).

By default, those changes are *unstaged*. The staging area is a layer between your working directory (where you make your modifications) and the local Git repository (the commits and all the metadata in the `.git` folder). To bring changes from the working directory to the Git repository, you need to *stage* them with the `git add` command, and then to commit them with the `git commit` command.

There are various commands you should know to review your current work, before staging it, while it is staged, and after it has been committed.

- `git diff` will show you the current unstaged changes, i.e. the differences between your working directory and the staging area.
- `git checkout -- <files>` will undo the unstaged changes to the given files.

- `git add <files>` will *stage* the changes on the listed files.
- `git diff --staged` will show the current staged changes, i.e. the differences between the staging area and the last commit.
- `git reset HEAD <files>` will *unstage* changes to the listed files.
- `git status` will show you what are the currently staged and unstaged modifications.
- `git commit` will commit the staged files. It will open a text editor (you can define the one you want to use with the `GIT_EDITOR` environment variable or the `core.editor` setting in your Git configuration) to let you write a commit log. You can use `git commit -m "Cool commit log"` to write the log directly.
- `git commit --amend` lets you amend the last commit with your currently staged changes (added with `git add`). This is the best option if you want to fix a mistake in the last commit (bug, typo, style issue, etc.).
- `git log` will show you the last commits of your current branch. If you did local commits, they should be shown at the top.
- `git show` will show you the changes of the last commit. You can also specify a commit hash to see the changes for that commit.

That's a lot to memorize! Don't worry, just check this cheat sheet when you need to make changes, and learn by doing.

Here's how the shell history could look like on our example:

```
# It's nice to know where you're starting from
git log

# Do changes to the Project Manager with the nano text editor
nano editor/project_manager.cpp

# Find an unrelated bug in Control and fix it
nano scene/gui/control.cpp

# Review changes
git status
git diff
```

```
# We'll do two commits for our unrelated changes,
# starting by the Control changes necessary for the PM
enhancements
git add scene/gui/control.cpp
git commit -m "Fix handling of margins in Control"

# Check we did good
git log
git show
git status

# Make our second commit
git add editor/project_manager.cpp
git commit -m "Add a pretty banner to the Project Manager"
git log
```

With this, we should have two new commits in our `better-project-manager` branch which were not in the `master` branch. They are still only local though, the remote fork does not know about them, nor does the upstream repo.

Pushing changes to a remote

That's where `git push` will come into play. In Git, a commit is always done in the local repository (unlike Subversion where a commit will modify the remote repository directly). You need to *push* the new commits to a remote branch to share them with the world. The syntax for this is:

```
git push <remote> <local branch>[:<remote branch>]
```

The part about the remote branch can be omitted if you want it to have the same name as the local branch, which is our case in this example, so we will do:

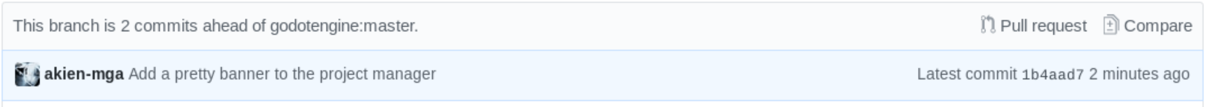
```
git push origin better-project-manager
```

Git will ask you for your username and password. For your password, enter your GitHub Personal Access Token (PAT). If you do not have a GitHub Personal Access Token, or do not have one with the correct permissions for your newly forked repository, you will need to create one. Follow this link to create your Personal Access Token: Creating a personal access token [https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token].

After you have successfully verified your account using your PAT, the changes will be sent to your remote repository. If you check the fork's page on GitHub, you should see a new branch with your added commits.

Issuing a pull request

When you load your fork's branch on GitHub, you should see a line saying *"This branch is 2 commits ahead of godotengine:master."* (and potentially some commits behind, if your `master` branch was out of sync with the upstream `master` branch).



This branch is 2 commits ahead of godotengine:master.

akien-mga Add a pretty banner to the project manager

Latest commit `1b4aad7` 2 minutes ago

On that line, there is a "Pull request" link. Clicking it will open a form that will let you issue a pull request on the `godotengine/godot` upstream repository. It should show you your two commits, and state "Able to merge". If not (e.g. it has way more commits, or says there are merge conflicts), don't create the PR yet, something went wrong. Go to our Godot Contributors Chat [https://chat.godotengine.org/] and ask for support :)

Use an explicit title for the PR and put the necessary details in the comment area. You can drag and drop screenshots, GIFs or zipped projects if relevant, to showcase what your work implements. Click "Create a pull request", and tadaa!

Modifying a pull request

While it is reviewed by other contributors, you will often need to make changes to your yet-unmerged PR, either because contributors requested them, or because you found issues yourself while testing.

The good news is that you can modify a pull request simply by acting on the branch you made the pull request from. You can e.g. make a new commit on that branch, push it to your fork, and the PR will be updated automatically:

```
# Check out your branch again if you had changed in the
meantime
git checkout better-project-manager

# Fix a mistake
nano editor/project_manager.cpp
git add editor/project_manager.cpp
git commit -m "Fix a typo in the banner's title"
git push origin better-project-manager
```

However, be aware that in our PR workflow, we favor commits that bring the codebase from one functional state to another functional state, without having intermediate commits fixing up bugs in your own code or style issues. Most of the time, we will prefer a single commit in a given PR (unless there's a good reason to keep the changes separate). Instead of authoring a new commit, consider using `git commit --amend` to amend the previous commit with your fixes. The above example would then become:

```
# Check out your branch again if you had changed in the
meantime
git checkout better-project-manager
```

```
# Fix a mistake
nano editor/project_manager.cpp
git add editor/project_manager.cpp
# --amend will change the previous commit, so you will have the
opportunity
# to edit its commit message if relevant.
git commit --amend
# As we modified the last commit, it no longer matches the one
from your
# remote branch, so we need to force push to overwrite that
branch.
git push --force origin better-project-manager
```

The interactive rebase

If you didn't follow the above steps closely to *amend* changes into a commit instead of creating fixup commits, or if you authored your changes without being aware of our workflow and Git usage tips, reviewers might request you to *rebase* your branch to *squash* some or all of the commits into one.

Indeed, if some commits have been made following reviews to fix bugs, typos, etc. in the original commit, they are not relevant to a future changelog reader who would want to know what happened in the Godot codebase, or when and how a given file was last modified.

To squash those extraneous commits into the main one, we will have to *rewrite history*. Right, we have that power. You may read that it's a bad practice, and it's true when it comes to branches of the upstream repo. But in your fork, you can do whatever you want, and everything is allowed to get neat PRs :)

We will use the *interactive rebase* `git rebase -i` to do this. This command takes a commit ID or a branch name as argument, and will let you modify all commits between that commit/branch and the last one in your working branch, the so-called `HEAD`.

While you can give any commit ID to `git rebase -i` and review everything in between, the most common and convenient workflow involves rebasing on the upstream `master` branch, which you can do with:

```
git rebase -i upstream/master
```

Note

Referencing branches in Git is a bit tricky due to the distinction between remote and local branches. Here, `upstream/master` (with a */*) is a local branch which has been pulled from the `upstream` remote's `master` branch.

Interactive rebases can only be done on local branches, so the */* is important here. As the upstream remote changes frequently, your local `upstream/master` branch may become outdated, so you can update it with `git fetch upstream master`. Contrarily to `git pull --rebase upstream master` which would update your currently checked out branch, `fetch` will only update the `upstream/master` reference (which is distinct from your local `master` branch... yes it's confusing, but you'll become familiar with this little by little).

This will open a text editor (`vi` by default, see Git docs [https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration#_core_editor] to configure your favorite one) with something which may look like this:

```
pick 1b4aad7 Add a pretty banner to the Project Manager
pick e07077e Fix a typo in the banner's title
```

The editor will also show instructions regarding how you can act on those commits. In particular, it should tell you that "pick" means to use that commit (do nothing), and that "squash" and "fixup" can be used to *meld* the commit in its parent commit. The difference between "squash" and "fixup" is that "fixup" will discard the commit log from the squashed commit. In our example, we are not interested in keeping the log of the "Fix a typo" commit, so we use:

```
pick 1b4aad7 Add a pretty banner to the Project Manager
fixup e07077e Fix a typo in the banner's title
```

Upon saving and quitting the editor, the rebase will occur. The second commit will be melded into the first one, and `git log` and `git show` should now confirm that you have only one commit with the changes from both previous commits.

But! You rewrote the history, and now your local and remote branches have diverged. Indeed, commit 1b4aad7 in the above example will have changed, and therefore got a new commit hash. If you try to push to your remote branch, it will raise an error:

```
git push origin better-project-manager
To https://github.com/akien-mga/godot
 ! [rejected]        better-project-manager -> better-project-
manager (non-fast-forward)
error: failed to push some refs to 'https://akien-
mga@github.com/akien-mga/godot'
hint: Updates were rejected because the tip of your current
branch is behind
hint: its remote counterpart.
```

This is reasonable behavior, Git will not let you push changes that would override remote content. But that's actually what we want to do here, so we will have to *force* it:

```
git push --force origin better-project-manager
```

And tadaa! Git will happily *replace* your remote branch with what you had locally (so make sure that's what you wanted, using `git log`). This will also update the PR accordingly.

Rebasing onto another branch

If you have accidentally opened your PR on the wrong branch, or need to target another branch for some reason, you might need to filter out a lot of commits that differ between the old branch (for example `4.2`) and the new branch (for example `master`). This can make rebasing difficult and tedious. Fortunately `git` has a command just for this situation, `git rebase --onto`.

If your PR was created from the `4.2` branch and you want to update it to instead start at `master` the following steps *should* fix this in one step:

```
git rebase -i --onto master 4.2
```

This will take all the commits on your branch *after* the `4.2` branch, and then splice them on top of `master`, ignoring any commits from the `4.2` branch not on the `master` branch. You may still need to do some fixing, but this command should save you a lot of tedious work removing commits.

Just like above for the interactive rebase you need to force push your branch to handle the different changes:

```
git push --force origin better-project-manager
```

Deleting a Git branch

After your pull request gets merged, there's one last thing you should do: delete your Git branch for the PR. There won't be issues if you don't delete your branch, but it's good practice to do so. You'll need to do this twice, once for the local branch and another for the remote branch on GitHub.

To delete our better Project Manager branch locally, use this command:

```
git branch -d better-project-manager
```

Alternatively, if the branch hadn't been merged yet and we wanted to delete it anyway, instead of `-d` you would use `-D`.

Next, to delete the remote branch on GitHub use this command:

```
git push origin -d better-project-manager
```

You can also delete the remote branch from the GitHub PR itself, a button should appear once it has been merged or closed.

Pull request review process

Note

This page is intended to provide insight into the pull request (PR) review process that we aspire to. As such, it is primarily targeted at engine maintainers who are responsible for reviewing and approving pull requests. That being said, much of the content is useful for prospective contributors wanting to know how to ensure that their PR is merged.

From a high level, the ideal life cycle of a pull request looks like the following:

1. A contributor opens a PR that fixes a specific problem (optimally closing a GitHub issue [https://github.com/godotengine/godot] or implementing a proposal [https://github.com/godotengine/godot-proposals]).
2. Other contributors provide feedback on the PR (including reviewing and/or approving the PR, as appropriate).
3. An engine maintainer reviews the code and provides feedback, requests changes, or approves the pull request, as appropriate.
4. Another maintainer reviews the code with a focus on code style/clarity and approves it once satisfied.
5. A team leader or a member of the production team [https://godotengine.org/teams#production] merges the pull request if satisfied that it has been sufficiently reviewed.

This document will explain steps 2, 3, 4, and 5 in more detail. For a more detailed explanation of the pull request workflow please see the pull request workflow document.

Note

In practice these steps may blend together. Oftentimes maintainers will provide comments on code style and code quality at the same time and will approve a pull request for both.

Typically the first interaction on a pull request will be an engine maintainer assigning tags to the pull request and flagging it for review by someone familiar with that area of code.

Engine maintainers are folks who are "members" of the Godot project repository on GitHub and/or are listed on the Teams page [https://godotengine.org/teams] on the Godot website. Maintainers are responsible for a given area of the engine. Typically this means they are the people who are given more trust to approve and recommend pull requests for merging.

Even if you are not a maintainer, you can still help by reviewing code, providing feedback on PRs and testing PRs locally on your machine to confirm that they work as intended. Many of the currently active maintainers started out doing this before they became maintainers.

Code review and testing

The following is a list of things that contributors and engine maintainers can do to conduct a substantive code review of a pull request.

Note

If you want to conduct a code review, but can't do everything on this list, say that in your review comment. For example, it is still very helpful to provide comments on code, even if you can't build the pull request locally to test the pull request (or vice versa). Feel free to review the code, just remember to make a note at the end of your review that you have reviewed the code only and have not tested the changes locally.

1. Confirm that the problem exists

PRs need to solve problems and problems need to be documented. Make sure that the pull request links and closes (or at least addresses) a bug or a proposal. If it doesn't, consider asking the contributor to update the opening message of the PR to explain the problem that the PR aims to solve in more detail.

Note

It should be clear _why_ a pull request is needed before it is merged. This assists reviewers in determining whether a PR does what it says it does and it helps contributors in the future understand why the code is the way it is.

2. Test the PR and look for regressions

While strict code review and CI help to ensure that all pull requests work as intended, mistakes happen and sometimes contributors push code that creates a problem in addition to solving a problem. Maintainers will avoid merging code that contains a regression even if it solves the problem as intended.

When reviewing a pull request, ensure that the PR does what it says it does (i.e. fixes the linked bug or implements the new feature) and nothing outside of the PR target area is broken by the change. You can do this by running the editor and trying out some common functions of the editor (adding objects to a scene, running GDScript, opening and closing menus etc.). Also, while reviewing the code, look for suspicious changes in other parts of the engine. Sometimes during rebasing changes slip through that contributors are not aware of.

3. Do a code review

Code reviews are usually done by people who are already experienced in a given area. They may be able to provide ideas to make code faster, more organized, or more idiomatic. But, even if you are not very experienced, you may want to conduct a code review to provide feedback within the scope of what you are comfortable reviewing. Doing so is valuable for the area maintainer (as a second set of eyes on a problem is always helpful) and it is also helpful for you as it will help you get more familiar with that area of code and will expose you to how other people solve problems. In fact, reviewing the code of experienced engine maintainers is a great way to get to know the codebase.

Here are some things to think about and look out for as you review the code:

- **Code only touches the areas announced in the PR (and the commit message).**

  It can be tempting to fix random things in the code, as you see them. However, this can quickly make a pull request difficult to review and can make it hard to dig through in the commit history. Small touch-ups next to the related area are alright, but often bugs that you can find along the way are better fixed in their own PRs.

- **Code properly uses Godot's own APIs and patterns.**

  Consistency is very important, and a solution that already exists in the codebase is preferable to an ad-hoc solution.

- **Are core areas affected by the change?**

  Sometimes a PR that is supposed to solve a local problem can have a far-reaching effect way outside of its scope. Usually it is best to keep code changes local to where the problem arises. If you think that the solution requires changes outside the scope of the problem, it is usually best to seek the opinion of a team leader who may have another idea for how to solve the problem.

4. Iterate with the contributor and improve the PR

Maintainers should provide feedback and suggestions for improvement if they spot things in the code that they would like changed. Preferably, suggestions should come in order of importance: first, address overall code design and the approach to solving the problem, then make sure the code is complying with the engine's best practices, and lastly, do the code style review.

Note

Communicate barriers to merging early in the review process.

If the PR has clear blockers or will likely not get merged for whatever other reason, that fact should be communicated as early and clearly as possible. We want to avoid stringing people along because it feels bad to say "sorry, no".

As you review pull requests, keep the Godot Code of Conduct [https://godotengine.org/code-of-conduct] in mind. Especially the following:

- Politeness is expected at all times. Be kind and courteous.

- Always assume positive intent from others.
- Feedback is always welcome but keep your criticism constructive.

Here are some things to avoid as you iterate on a pull request with a contributor:

- **Needless double reviews.**

  In other words, review the full PR at once and avoid coming back endless times to point out issues that you could have noted in the first review. Of course, this can't always be avoided, but we should try to catch everything at once.

- **Being overly nitpicky.**

  Code quality can be flexible depending on the area of the engine you are working in. In general, our standard for code quality is much higher in core areas and in performance-sensitive areas than it is in editor code for example.

- **Expanding the scope of a pull request.**

  Providing context or related/similar issues or proposals that may be fixed similarly can be helpful, but adding a "may as well fix that thing over there as well while at it" or "could we add to this as well?" isn't always fair to the contributor. Use your judgement when deciding whether additional fixes are within scope, but try to keep the scope as close to the original pull request as possible.

And ultimately, don't feel pressured to deal with the PR all alone. Feel free to ask for a helping hand on the Godot Contributors Chat [https://chat.godotengine.org], in the appropriate channel or in #general. Other teams may already be tagged for review, so you can also wait or ask for their assistance.

5. Approve the pull request

After reviewing the code, if you think that the code is ready to be merged into the engine, then go ahead and "approve" it. Make sure to also comment and specify the nature of your review (i.e. say whether you ran the code locally, whether you reviewed for style as well as correctness, etc.). Even if you are not an engine maintainer, approving a pull request signals to others that the code is good and likely solves the problem the PR says it does. Approving a pull request as a non-engine maintainer does not guarantee that the code will be merged, other people will still review it, so don't be shy.

Code style review

Generally speaking, we aim to conduct a code review before a style/clarity review as contributors typically want to know if their general approach is acceptable before putting in the effort to make nitpicky changes to style. In other words, maintainers shouldn't ask contributors to change the style of code that may need to be rewritten in subsequent reviews. Similarly, maintainers should avoid asking for contributors to rebase PRs if the PR has not been reviewed.

That being said, not everyone feels confident enough to provide a review on code correctness, in that case, providing comments on code style and clarity ahead of a more substantive code review is totally appropriate and more than welcome.

In practice the code style review can be done as part of the substantive code review. The important thing is that both the substantive code and the code style need to be reviewed and considered before a pull request is merged.

When reviewing code style pay particular attention to ensuring that the pull request follows the Code style guidelines. While `clang-`

`format` and various CI checks can catch a lot of inconsistencies, they are far from perfect and are unable to detect some issues. For example, you should check that:

- The style of header includes is respected.
- Identifiers use `snake_case` and follow our naming conventions.
- Method parameters start with `p_*` or `r_*` (if they are used to return a value).
- Braces are used appropriately, even for one-liner conditionals.
- Code is properly spaced (exactly one empty line between methods, no unnecessary empty lines inside of method bodies).

Note

This list is not complete and doesn't aim to be complete. Refer to the linked style guide document for a complete set of rules. Keep in mind that `clang-format` may not catch things you hope it would, so pay attention and try to build a sense of what exactly it can and cannot detect.

Merging pull requests

In general, pull requests should only be merged by members of the production team or team leaders for pull requests in their area of the engine. For example, the networking team leader could merge a networking pull request that doesn't substantially change non-networking sections of code.

In practice it is best to wait for a member of the production team to merge the pull request as they keep a close eye on the entire codebase and will likely have a better sense of what other recent/upcoming changes this pull request may conflict with (or any other reason that it may make sense to delay the pull request). Feel free to leave a comment saying that the PR should be ready to merge.

The following are the steps to take before merging a pull request. The degree to which you adhere to these steps can be flexible for simple/straightforward pull requests, but they should be carefully taken for complex or risky pull requests.

As a contributor you can help move a pull request forward by doing some of these steps yourself.

1. Get feedback from the right people/teams

Production team members should ensure that the right people look at a pull request before it is merged. In some cases this may require multiple people to weigh in. In other cases, only one substantive approval is needed before the code can be merged.

In general, try not to merge things based on one review alone, especially if it is your own. Get a second opinion from another maintainer, and make sure all the teams that may be impacted have been reasonably represented by the reviewers. For example, if a pull request adds to the documentation, it's often useful to let the area maintainers check it for factual correctness and let documentation maintainers check it for formatting, style, and grammar.

A good rule of thumb is that at least one subject matter expert should have approved the pull request for correctness, and at least one other maintainer should have approved the pull request for code style. Either of those people could be the person merging the pull request.

Make sure that the reviews and approvals were left by people competent in that specific engine area. It is possible that even a long-standing member of the Godot organization left a review without having the relevant expertise.

Note

An easy way to find PRs that may be ready for merging is filtering by approved PRs and sorting by recently updated. For example, in the main Godot repository, you can use this link [https://github.com/godotengine/godot/pulls?q=is%3Apr+is%3Aopen+review%3Aapproved+sort%3Aupdated-desc].

2. Get feedback from the community

If a pull request is having trouble attracting reviewers, you may need to reach out more broadly to ask for help reviewing. Consider asking:

- the person who reported the bug if the pull request fixes the bug for them,
- contributors who have recently edited that file if they could take a look, or
- a more experienced maintainer from another area if they could provide feedback.

3. Git checklist

- **Make sure that the PR comes in one commit.**

  When each commit is self-contained and could be used to build a clean and working version of the engine, it may be okay to merge a pull request with multiple commits, but in general, we require that all pull requests only have one commit. This helps us keep the Git history clean.

- **Fixes made during the review process must be squashed into the main commit.**

  For multi-commit PRs check that those fixes are amended in the relevant commits, and are not just applied on top of everything.

- **Make sure that the PR has no merge conflicts.**

  Contributors may need to rebase their changes on top of the relevant branch (e.g. `master` or `3.x`) and manually fix merge conflicts. Even if there are no merge conflicts, contributors may need to rebase especially old PRs as the GitHub conflict checker may not catch all conflicts, or the CI may have changed since it was originally run.

- **Check for proper commit attribution.**

  If a contributor uses an author signature that is not listed in their GitHub account, GitHub won't link the merged pull request to their account. This keeps them from getting proper credit in the GitHub history and makes them appear like a new contributor on the GitHub UI even after several contributions.

  Ultimately, it's up to the user if they want to fix it, but they can do so by authoring the Git commit with the same email they use for their GitHub account, or by adding the email they used for the Git commit to their GitHub profile.

- **Check for proper commit messages.**

  While we don't have a very strict ruleset for commit messages, we still require them to be short yet descriptive and use proper English. As a maintainer you've probably written them enough times to know how to make one, but for a general template think about *"Fix <issue> in <part of codebase>"*. For a more detailed recommendation see the contributing.md [https://github.com/godotengine/godot/blob/master/CONTRIBUTING.md#for mat-your-commit-messages-with-readability-in-mind] page in the main Godot repository.

4. GitHub checklist

- **Validate the target branch of the PR.**

  Most Godot development happens around in the `master` branch. Therefore most pull requests must be made against it. From there pull requests can then be backported to other branches. Be wary of people making PRs on the version they are using (e.g, `3.3`) and guide them to make a change against a higher-order branch (e.g. `3.x`). If the change is not applicable for the `master` branch, the initial PR can be made against the current maintenance branch, such as `3.x`. It's okay for people to make multiple PRs for each target branch, especially if the changes cannot be easily backported. Cherry-picking is also an option, if possible. Use the appropriate labels if the PR can be cherrypicked (e.g. `cherrypick:3.x`).

Note

It is possible to change the target branch of the PR, that has already been submitted, but be aware of the consequences. As it cannot be synchronized with the push, the target branch change will inevitable tag the entire list of maintainers for review. It may also render the CI incapable of running properly. A push should help with that, but if nothing else, recommend opening a new, fresh PR.

- **Make sure that the appropriate milestone is assigned.**

  This will make it more obvious which version would include the submitted changes, should the pull request be merged now.

Note, that the milestone is not a binding contract and does not guarantee that this version is definitely going to include the PR. If the pull request is not merged before the version is released, the milestone will be moved (and the PR itself may require a target branch change).

Similarly, when merging a PR with a higher milestone than the current version, or a "wildcard" milestone (e.g. "4.x"), ensure to update the milestone to the current version.

- **Make sure that the opening message of the PR contains the magic words "Closes #..." or "Fixes #...".**

  These link the PR and the referenced issue together and allow GitHub to auto-close the latter when you merge the changes. Note, that this only works for the PRs that target the `master` branch. For others you need to pay attention and close the related issues manually. Do it with *"Fixed by #..."* or *"Resolved by #..."* comment to clearly indicate the act for future contributors.

- **For the issues that get closed by the PR add the closest relevant milestone.**

  In other words, if the PR is targeting the `master` branch, but is then also cherrypicked for `3.x`, the next `3.x` release would be the appropriate milestone for the closed issue.

5. Merge the pull request

If it is appropriate for you to be merging a pull request (i.e. you are on the production team or you are the team leader for that area), you are confident that the pull request has been sufficiently reviewed, and once you carry out these steps you can go ahead and merge the pull request.

Testing pull requests

Many people are developing new features or fixing bugs on GitHub. To help with engine development, you may be asked to test those pull requests with a Godot build that includes code from the pull request in question.

Thanks to GitHub Actions, all pull requests [https://github.com/godotengine/godot/pulls] have continuous builds available. These builds let you try out pull requests without having to compile anything from source.

Downloading a compiled build

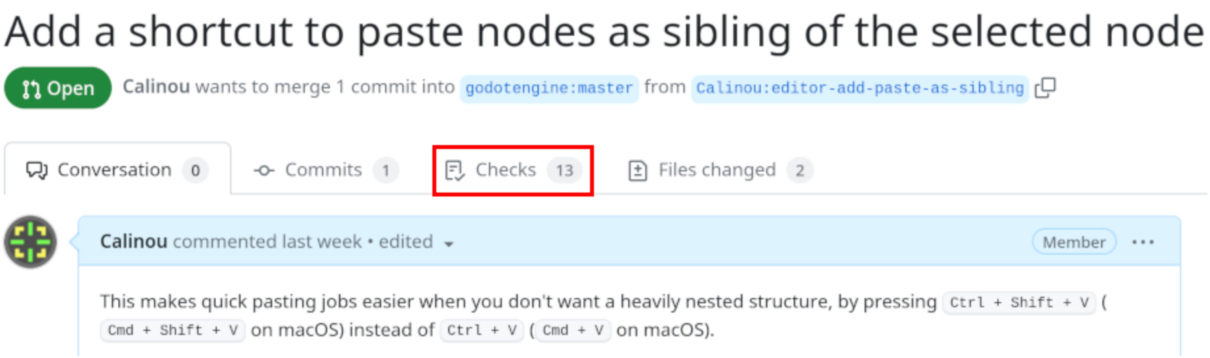
You can download pull request builds from GitHub Actions. Since only signed in users may download builds directly from GitHub Actions, the procedure varies depending on whether you have a GitHub account or not.

Note

Due to a GitHub Actions limitation, builds are only available for 90 days after the pull request was last updated. If you still wish to try a pull request locally, you can compile the pull request branch from source instead.

If you have a GitHub account

- Open the pull request page. Click the **Checks** tab near the top of the page:



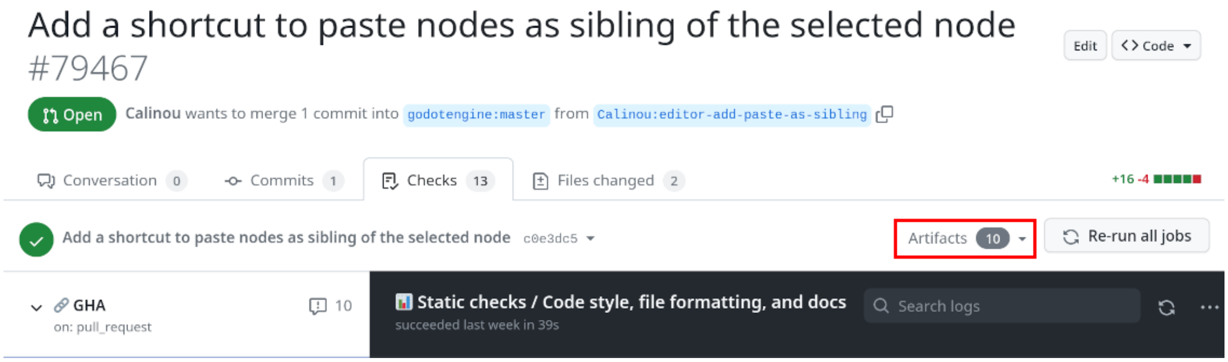
Add a shortcut to paste nodes as sibling of the selected node

Open Calinou wants to merge 1 commit into `godotengine:master` from `Calinou:editor-add-paste-as-sibling`

Conversation 0
Commits 1
Checks 13
Files changed 2
Calinou commented last week • edited
Member

This makes quick pasting jobs easier when you don't want a heavily nested structure, by pressing `Ctrl + Shift + V` (`Cmd + Shift + V` on macOS) instead of `Ctrl + V` (`Cmd + V` on macOS).

- Click the **Artifacts** dropdown on the right of the page:

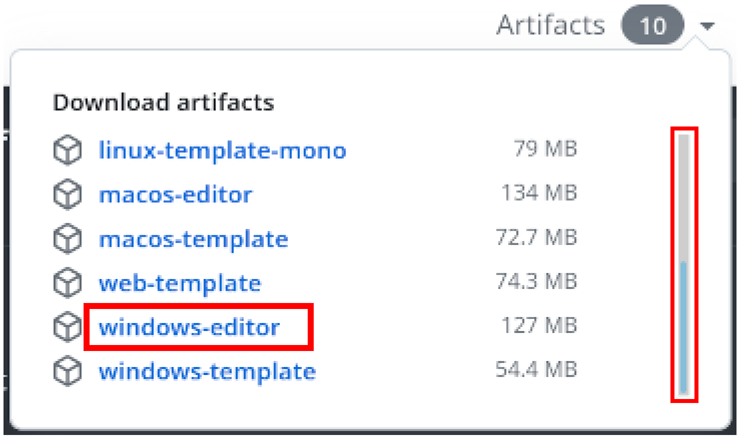


Add a shortcut to paste nodes as sibling of the selected node #79467

Open Calinou wants to merge 1 commit into `godotengine:master` from `Calinou:editor-add-paste-as-sibling`

Edit
Code
Conversation 0
Commits 1
Checks 13
Files changed 2
+16 -4
Add a shortcut to paste nodes as sibling of the selected node c0e3dc5
Artifacts 10
Re-run all jobs
GHA
on: pull_request
10
Static checks / Code style, file formatting, and docs
succeeded last week in 39s
Search logs

- In the dropdown, click the artifact's name to download it. Remember to scroll if you cannot see the name of the platform you're looking for:



Artifacts 10

Download artifacts

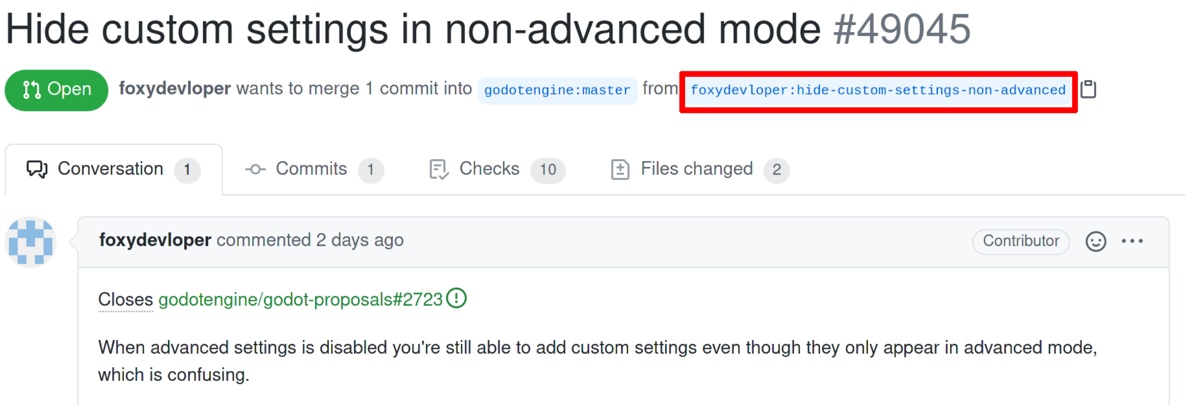
linux-template-mono	79 MB
macos-editor	134 MB
macos-template	72.7 MB
web-template	74.3 MB
windows-editor	127 MB
windows-template	54.4 MB

- Extract the ZIP archive then run the executable. Note that Windows and macOS binaries are **not** code signed. This means you may have to bypass a security warning before you can run the executable. On Windows, if you frequently test pull request builds, it may be better to disable Windows SmartScreen permanently in the Windows security settings. On macOS, see Running Godot apps on macOS for instructions on bypassing Gatekeeper.

If you don't have a GitHub account

If you don't have a GitHub account and can't sign up for one, you can use the third-party nightly.link [https://nightly.link] service to generate a universal download link.

- Open the pull request page. Click the *fork*'s branch name near the top of the page:



Hide custom settings in non-advanced mode #49045

Open foxydevloper wants to merge 1 commit into `godotengine:master` from `foxydevloper:hide-custom-settings-non-advanced`

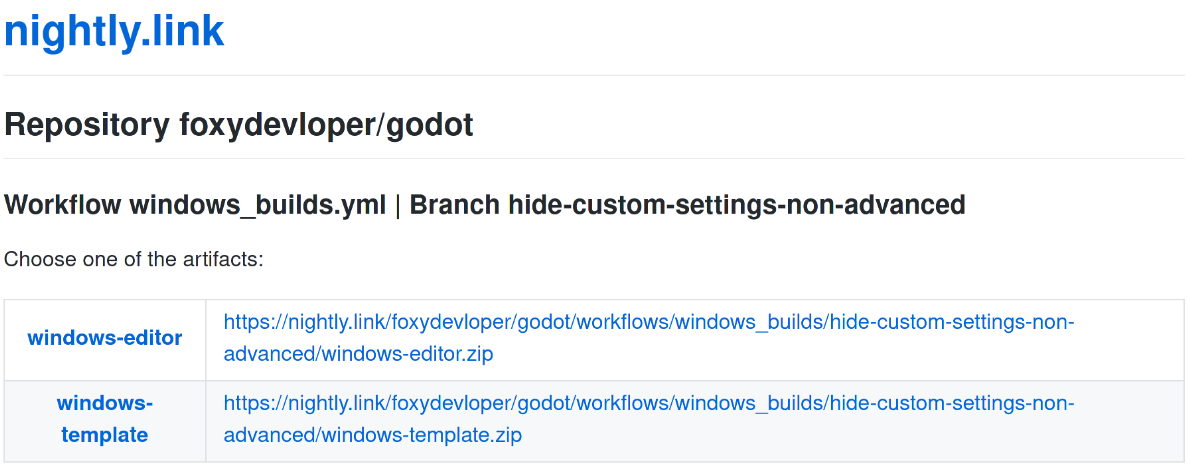
foxydevloper commented 2 days ago

Closes godotengine/godot-proposals#2723

When advanced settings is disabled you're still able to add custom settings even though they only appear in advanced mode, which is confusing.

- Now that you are on the fork's branch page, click the `.github` folder at the top of the file list. Then, click on the `workflows` folder (which is inside the `.github` folder). Click the workflow file for the platform you wish to download artifacts for. *After* clicking on the file (which opens the file view), copy the page URL from your browser's address bar.
- Open the nightly.link [https://nightly.link] website and paste the URL you just copied into the text field located below the heading **Paste a GitHub link, get a nightly.link!**. After

pasting the URL, click **Get links** on the right. If the format of the URL you pasted is correct, you should be presented with a page like this:



nightly.link

Repository foxydevloper/godot

Workflow windows_builds.yml | Branch hide-custom-settings-non-advanced

Choose one of the artifacts:

windows-editor	https://nightly.link/foxydevloper/godot/workflows/windows_builds/hide-custom-settings-non-advanced/windows-editor.zip
windows-template	https://nightly.link/foxydevloper/godot/workflows/windows_builds/hide-custom-settings-non-advanced/windows-template.zip

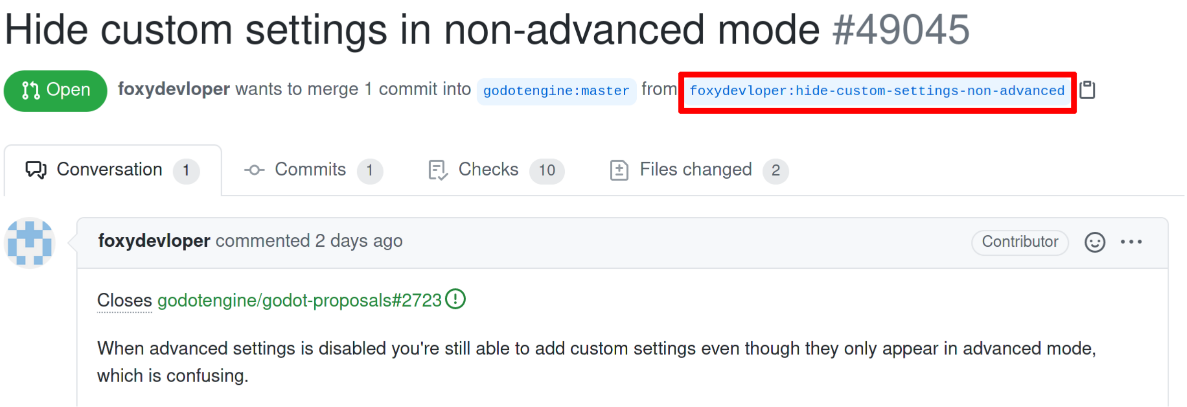
- Click the URL of the artifact you wish to download.
- Extract the ZIP archive then run the executable. Note that Windows and macOS binaries are not code signed. This means you may have to bypass a security warning before you can run the executable. If you frequently test pull request builds, it may be better to disable Windows SmartScreen or [disable macOS Gatekeeper](https://disable-gatekeeper.github.io/) [https://disable-gatekeeper.github.io/] permanently.

Compiling a pull request branch from source

This approach may be needed for pull requests that were last updated more than 90 days ago, or to test on platforms and configurations that are not supported by Godot's GitHub Actions setup.

Downloading a zipped pull request branch

- Open the pull request page. Click the *fork*'s branch name near the top of the page:



Hide custom settings in non-advanced mode #49045

Open **foxydevloper** wants to merge 1 commit into `godotengine:master` from `foxydevloper:hide-custom-settings-non-advanced`

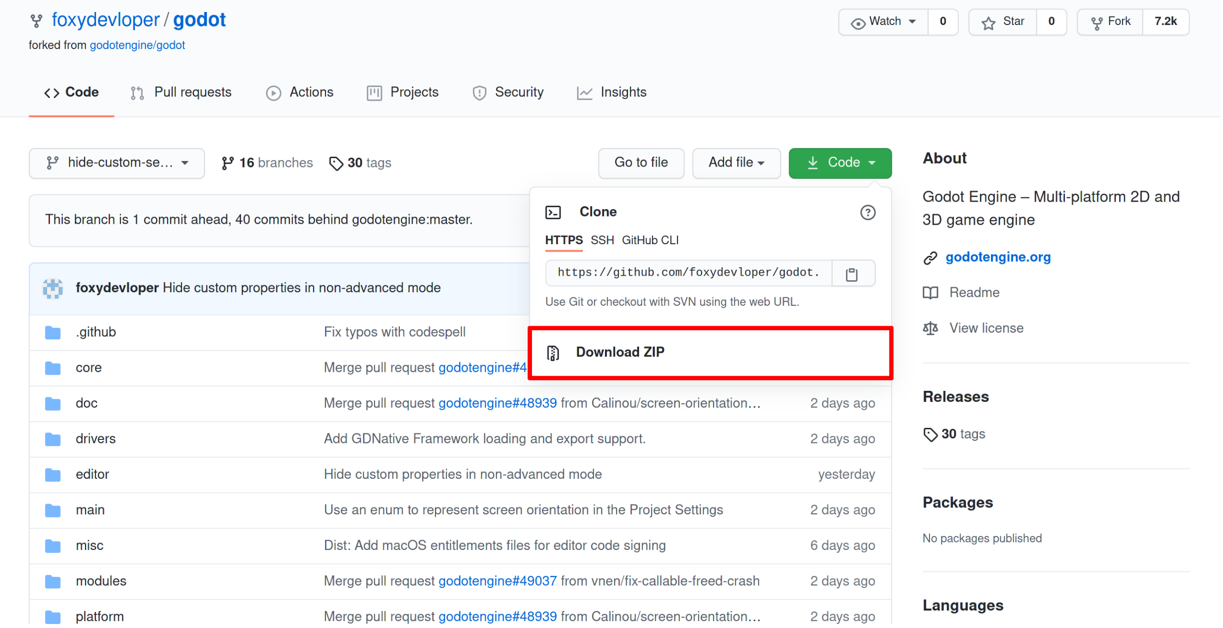
Conversation 1
Commits 1
Checks 10
Files changed 2
Contributor

foxydevloper commented 2 days ago

Closes godotengine/godot-proposals#2723

When advanced settings is disabled you're still able to add custom settings even though they only appear in advanced mode, which is confusing.

- Now that you are on the fork's branch page, click the green **Code** button on the right of the page then choose **Download ZIP** in the dropdown:



foxydevloper / **godot**

forked from godotengine/godot

Watch 0
Star 0
Fork 7.2k
Code
Pull requests
Actions
Projects
Security
Insights
hide-custom-se...
16 branches
30 tags
Go to file
Add file
Code
Clone
HTTPS
SSH
GitHub CLI
Download ZIP
About
Readme
View license
Releases
30 tags
Packages
Languages

This branch is 1 commit ahead, 40 commits behind godotengine:master.

`https://github.com/foxydevloper/godot.`

Use Git or checkout with SVN using the web URL.

Godot Engine – Multi-platform 2D and 3D game engine

godotengine.org

No packages published

foxydevloper Hide custom properties in non-advanced mode

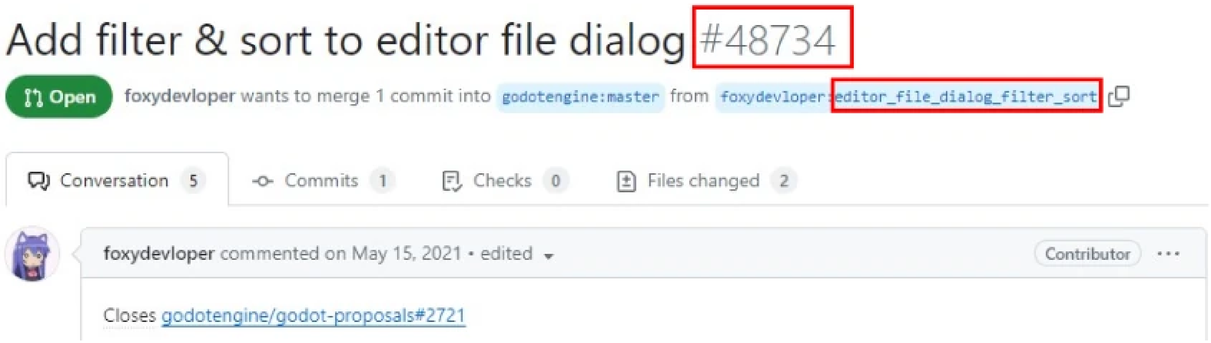
.github	Fix typos with codespell	
core	Merge pull request godotengine#4	
doc	Merge pull request godotengine#48939 from Calinou/screen-orientation…	2 days ago
drivers	Add GDNative Framework loading and export support.	2 days ago
editor	Hide custom properties in non-advanced mode	yesterday
main	Use an enum to represent screen orientation in the Project Settings	2 days ago
misc	Dist: Add macOS entitlements files for editor code signing	6 days ago
modules	Merge pull request godotengine#49037 from vnen/fix-callable-freed-crash	2 days ago
platform	Merge pull request godotengine#48939 from Calinou/screen-orientation…	2 days ago

- Extract the ZIP archive and follow the compiling instructions for your operating system.

Checking out a pull request branch with git

Alternatively, you can checkout the pull request directly with git:

- Open the pull request page. Note the pull request *number* (`PR_NUMBER`), and the *branch name* (`BRANCH_NAME`), but without the user name.



Add filter & sort to editor file dialog #48734

Open foxydevloper wants to merge 1 commit into godotengine:master from foxydevloper:editor_file_dialog_filter_sort

foxydevloper commented on May 15, 2021 • edited

Contributor

Closes godotengine/godot-proposals#2721

- Construct the command using this pattern:

```
git fetch upstream pull/PR_NUMBER/head:BRANCH_NAME
```

So for the pull request above, the actual command will be:

```
# Fetch PR branch locally
git fetch upstream
pull/48734/head:editor_file_dialog_filter_sort
```

- Once the pull request finishes downloading, checkout its branch:

```
git checkout editor_file_dialog_filter_sort
```

- And follow the compiling instructions for your operating system.

Engine development

The guides below explain how to contribute to the engine's codebase. You will learn about best practices when working on solutions and features, the code style and the internal structure of the codebase, how to set up your working environment, and more.

Be sure to also check the workflow guidelines, especially if you're new to using Git or GitHub.

- Best practices for engine contributors
- Code style guidelines
- C++ usage guidelines
- Handling compatibility breakages

Buildsystem and work environment

- Configuring an IDE
- Building from source
- Debugging and profiling

Engine architecture

- Engine core and modules
- Godot file formats
- Editor development

Best practices for engine contributors

Introduction

Godot has a large amount of users who have the ability to contribute because the project itself is aimed mainly at users who can code. That being said, not all of them have the same level of experience working in large projects or in software engineering, which can lead to common misunderstandings and bad practices during the process of contributing code to the project.

Language

The scope of this document is to be a list of best practices for contributors to follow, as well as to create a language they can use to refer to common situations that arise in the process of submitting their contributions.

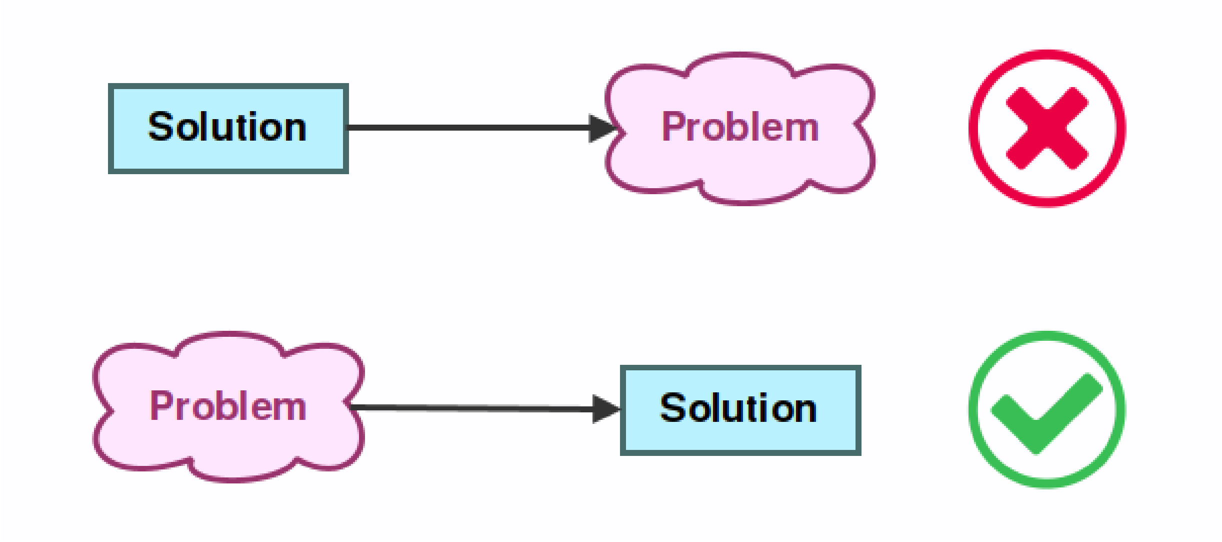
While a generalized list of software development best practices might be useful, we'll focus on the situations that are most common in our project.

Contributions are most of the time categorized as bug fixes, enhancements or new features. To abstract this idea, we will call them *Solutions*, because they always seek to solve something that can be described as a *Problem*.

Best Practices

#1: The problem always comes first

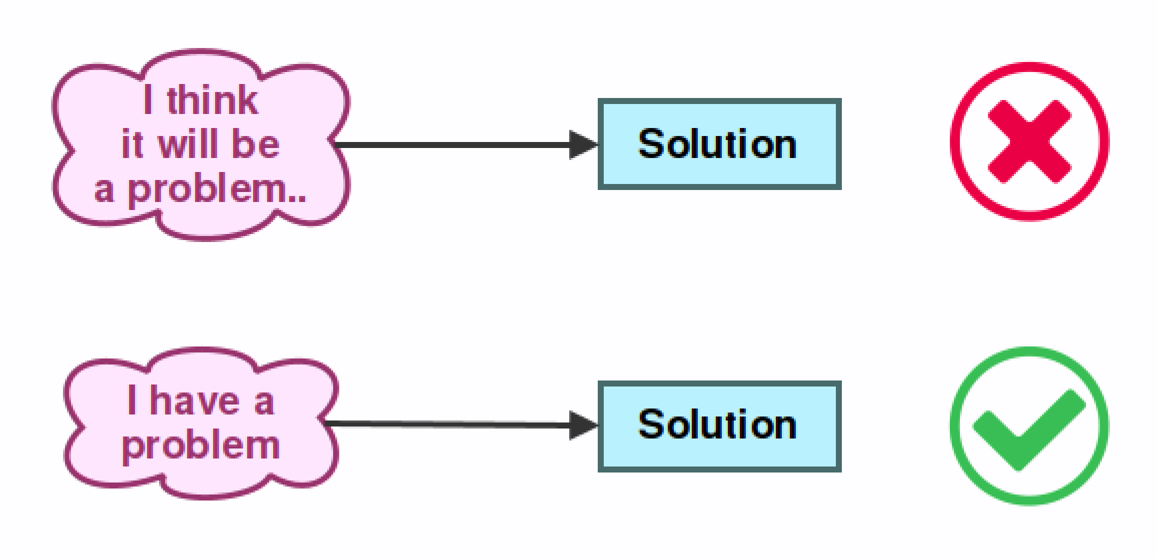
Many contributors are extremely creative and just enjoy the process of designing abstract data structures, creating nice user interfaces, or simply love programming. Whatever the case may be, they come up with cool ideas, which may or may not solve real problems.



These are usually called *solutions in search of a problem*. In an ideal world, they would not be harmful but, in reality, code takes time to write, takes up space and requires maintenance once it exists. Avoiding the addition of anything unnecessary is always considered a good practice in software development.

#2: To solve the problem, it has to exist in the first place

This is a variation of the previous practice. Adding anything unnecessary is not a good idea, but what constitutes what is necessary and what isn't?



The answer to this question is that the problem needs to *exist* before it can be actually solved. It must not be speculation or a belief. The user must be using the software as intended to create something they *need*. In this process, the user may stumble upon a problem that requires a solution to proceed, or in order to achieve greater productivity. In this case, *a solution is needed*.

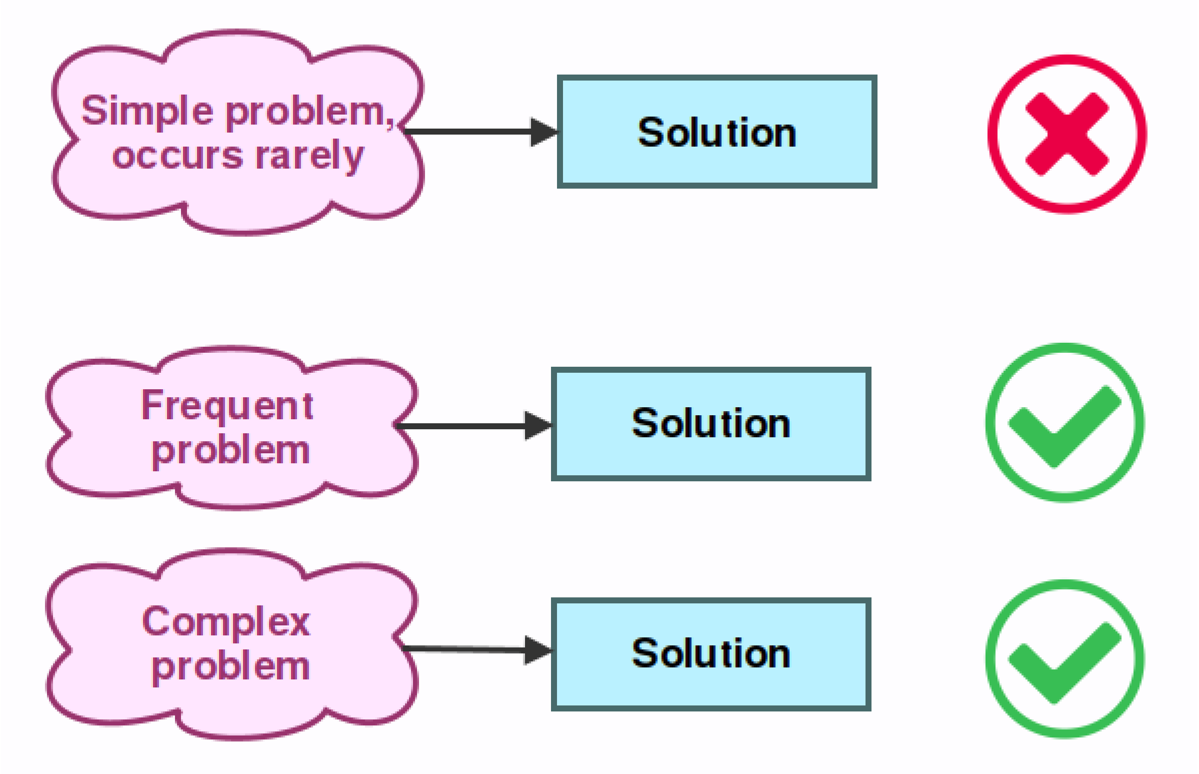
Believing that problems may arise in the future and that the software needs to be ready to solve them by the time they appear is called *"Future proofing"* and its characterized by lines of thought such as:

- I think it would be useful for users to...
- I think users will eventually need to...

This is generally considered a bad habit because trying to solve problems that *don't actually exist* in the present will often lead to code that will be written but never used, or that is considerably more complex to use and maintain than it needs to be.

#3: The problem has to be complex or frequent

Software is designed to solve problems, but we can't expect it to solve *every problem that exists under the sun*. As a game engine, Godot will help you make games better and faster, but it won't make an *entire game* for you. A line must be drawn somewhere.


Simple problem, occurs rarely
Solution
Frequent problem
Solution
Complex problem
Solution

Whether a problem is worth solving is determined by the effort that is required to work around it. The required effort depends on:

- The complexity of the problem
- The frequency the problem

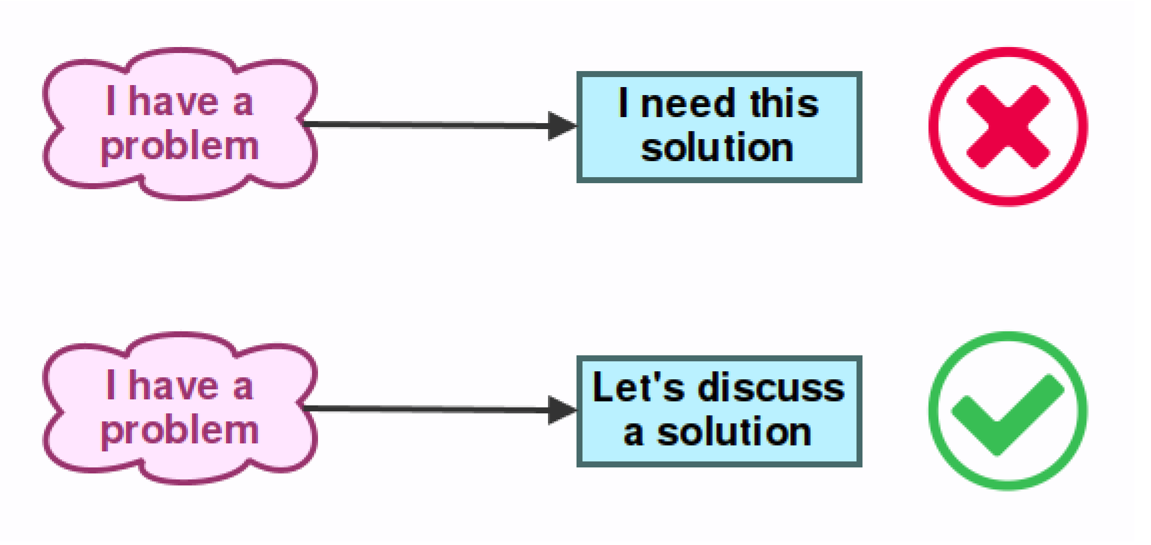
If the problem is *too complex* for most users to solve, then the software should offer a ready-made solution for it. Likewise, if the problem is easy for the user to work around, offering such a solution is unnecessary.

The exception, however, is when the user encounters a problem *frequently enough* that having to do the simple solution every time becomes an annoyance. In this case, the software should offer a solution to simplify the use case.

It's usually easy to tell if a problem is complex or frequent, but it can be difficult. This is why discussing with other developers (next point) is always advised.

#4: The solution must be discussed with others

Often, users will be immersed in their own projects when they stumble upon problems. These users will naturally try to solve the problem from their perspective, thinking only about their own use case. As a result, user proposed solutions don't always contemplate all use cases and are often biased towards the user's own requirements.



For developers, the perspective is different. They may find the user's problem too unique to justify a solution (instead of a workaround), or they might suggest a partial (usually simpler or lower level) solution that applies to a wider range of known problems and leave the rest of the solution up to the user.

In any case, before attempting to contribute, it is important to discuss the actual problems with the other developers or contributors, so a better agreement on implementation can be reached.

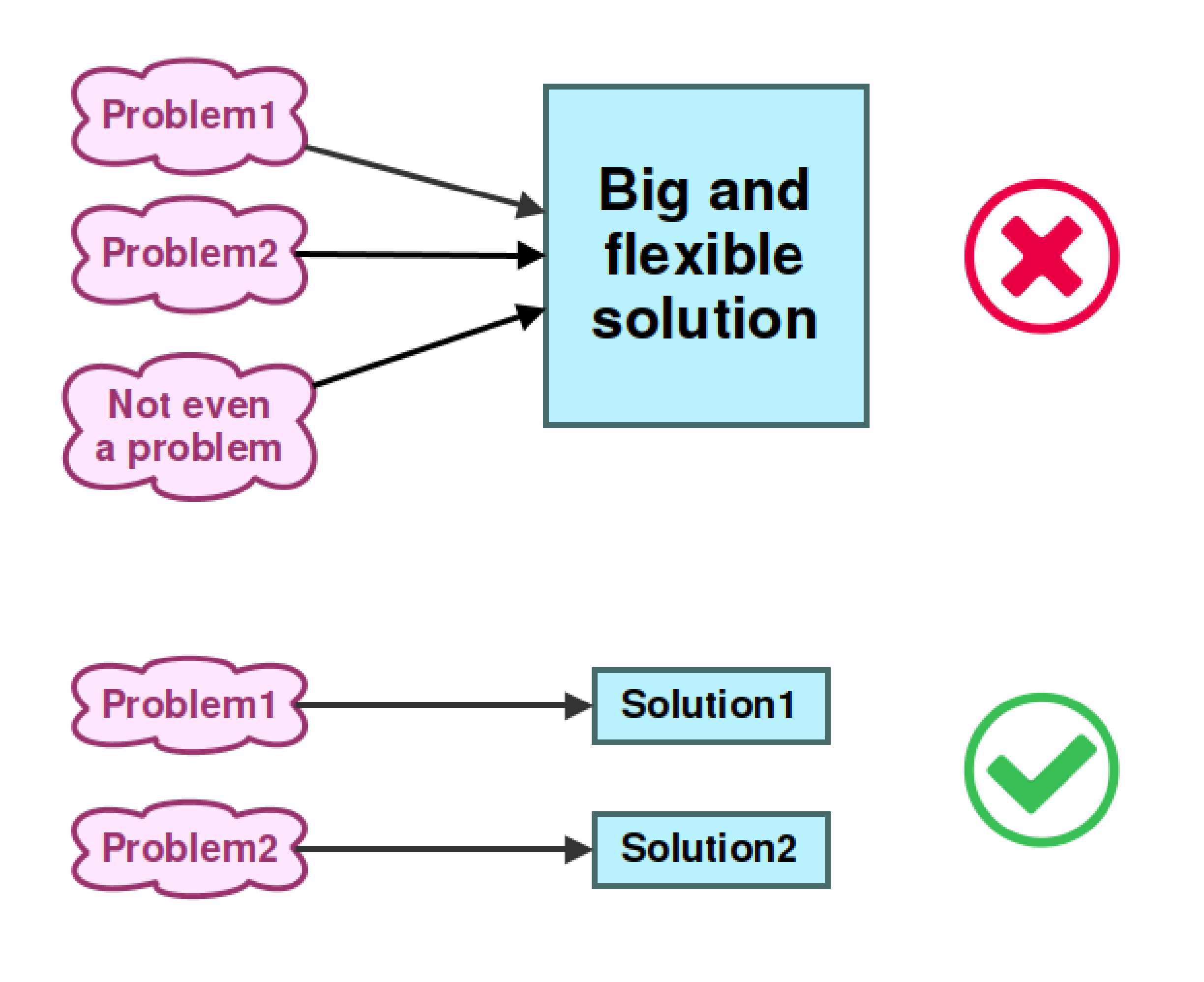
The only exception is when an area of code has a clear agreed upon owner, who talks to users directly and has the most knowledge to implement a solution directly.

Also, Godot's philosophy is to favor ease of use and maintenance over absolute performance. Performance optimizations will be considered, but they may not be accepted if they make something too difficult to use or if they add too much complexity to the codebase.

#5: To each problem, its own solution

For programmers, it is always a most enjoyable challenge to find the most optimal solutions to problems. It is possible to go overboard, though. Sometimes, contributors will try to come up with solutions that solve as many problems as possible.

The situation will often take a turn for the worse when, in order to make this solution appear even more fantastic and flexible, the pure speculation-based problems (as described in #2) also make their appearance on stage.


Problem1
Problem2
Not even a problem
Big and flexible solution
Problem1
Solution1
Problem2
Solution2

The main problem is that, in reality, it rarely works this way. Most of the time, writing an individual solution to each problem results in code that is simpler and more maintainable.

Additionally, solutions that target individual problems are better for the users. Targeted solutions allow users find something that does exactly what they need, without having to learn a more complex system they will only need for simple tasks.

Big and flexible solutions also have an additional drawback which is that, over time, they are rarely flexible enough for all users. Users end up requesting more and more functionality which ends up making the API and codebase more and more complex.

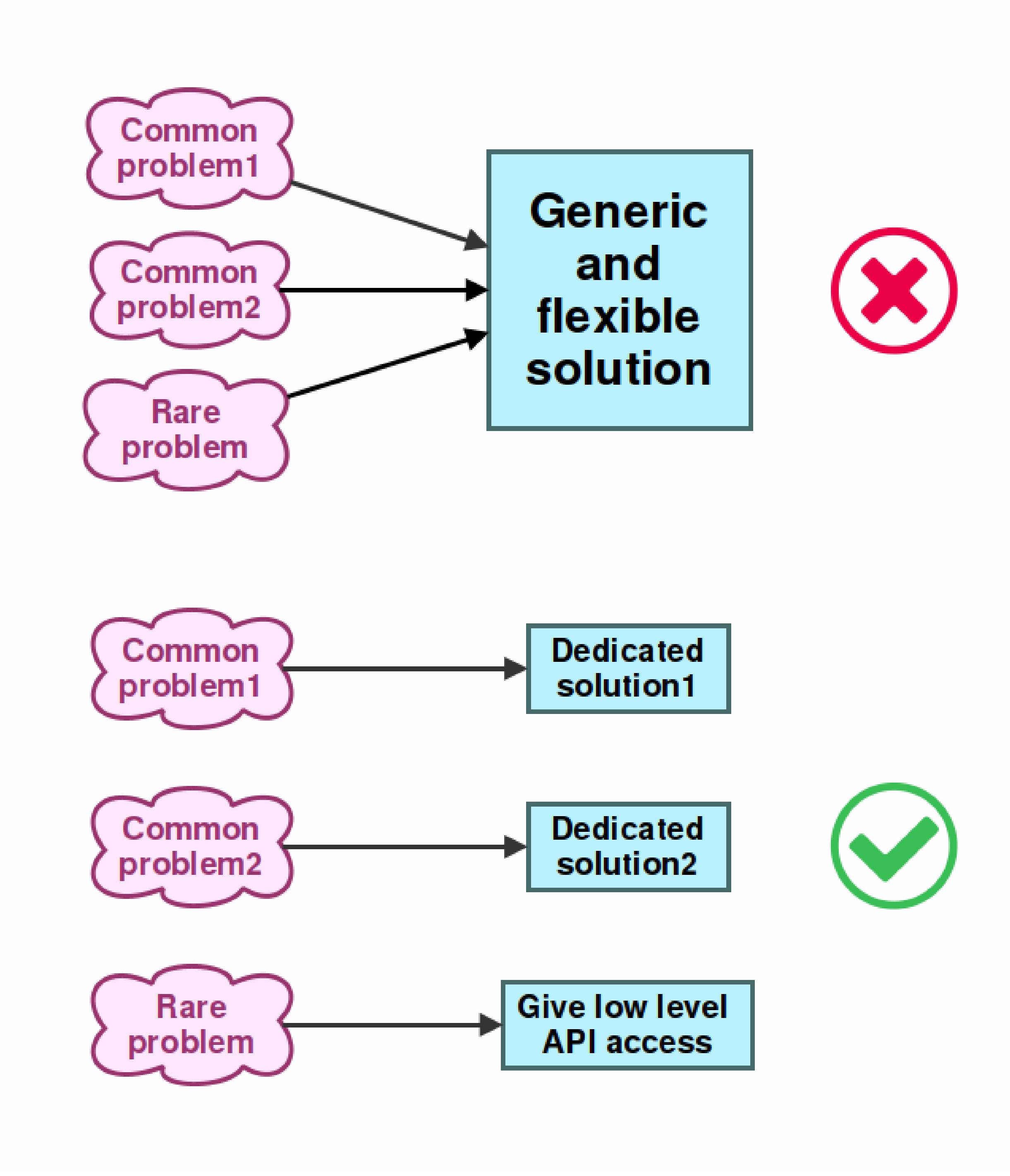
#6: Cater to common use cases, leave the door open for the rare ones

This is a continuation of the previous point, which further explains why this way of thinking and designing software is preferred.

As mentioned before (in point #2), it is very difficult for us (as human beings who design software) to actually understand all future user needs. Trying to write very flexible structures that cater to many use cases at once is often a mistake.

We may come up with something we believe is brilliant but later find out that users will never even use half of it or that they require features that don't quite fit into our original design, forcing us to either throw it away or make it even more complex.

The question is then, how do we design software that both allows users to do *what we know they need to do* now and allows them to do *what we don't yet know they'll need to do* in the future?


Common problem1
Common problem2
Rare problem
Generic and flexible solution
Common problem1
Dedicated solution1
Common problem2
Dedicated solution2
Rare problem
Give low level API access

The answer to this question is that, to ensure users still can do what they want to do, we need to give them access to a *low level API* that they can use to achieve what they want, even if it's more work for them because it means reimplementing some logic that already exists.

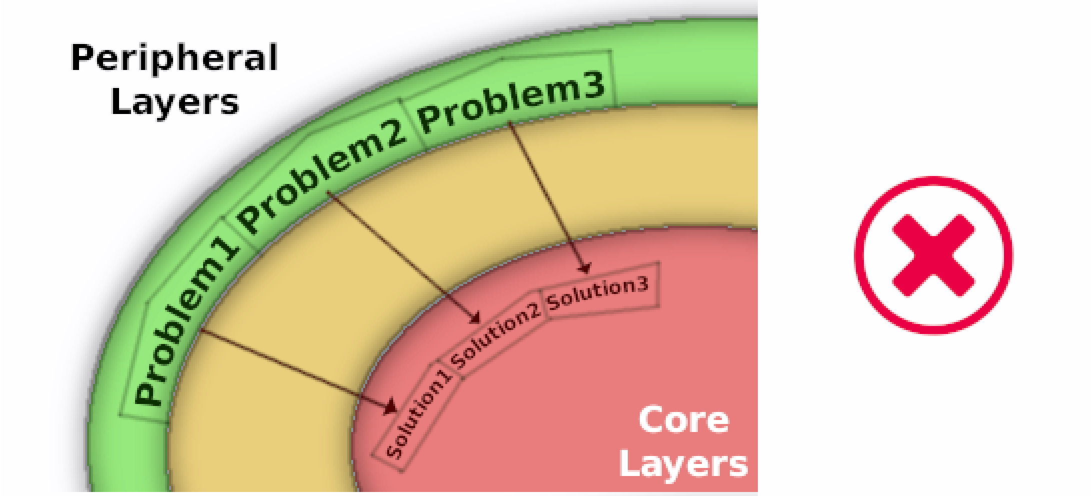
In real-life scenarios, these use cases will be at most rare and uncommon anyway, so it makes sense a custom solution needs to be written. This is why it's important to still provide users the basic building blocks to do it.

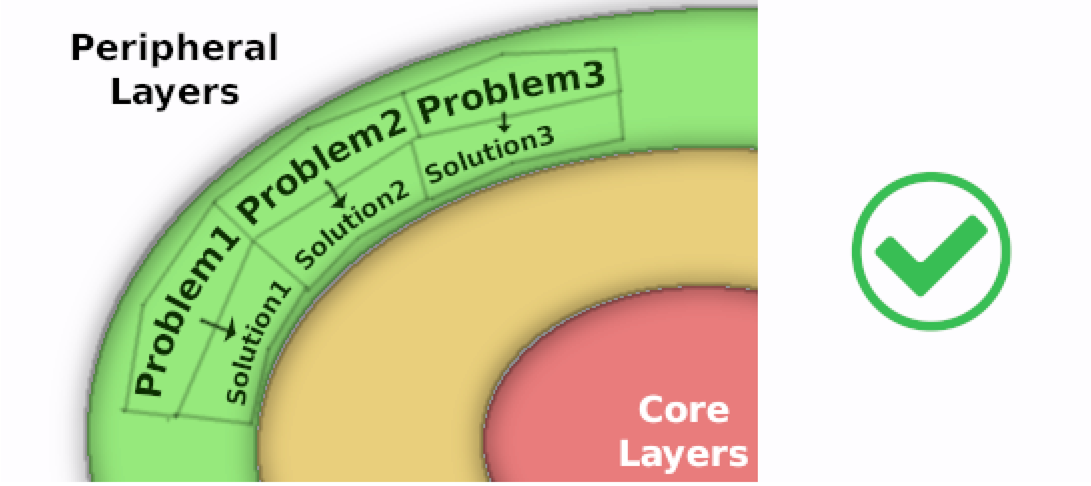
#7: Prefer local solutions

When looking for a solution to a problem, be it implementing a new feature or fixing a bug, sometimes the easiest path is to add data or a new function in the core layers of code.

The main problem here is, adding something to the core layers that will only be used from a single location far away will not only make the code more difficult to follow (split in two), but also make the core API larger, more complex, more difficult to understand in general.

This is bad, because readability and cleanness of core APIs is always of extreme importance given how much code relies on it, and because it's key for new contributors as a starting point to learning the codebase.


Peripheral Layers
Problem1
Problem2
Problem3
Solution1
Solution2
Solution3
Core Layers


Peripheral Layers
Problem1
Problem2
Problem3
Solution1
Solution2
Solution3
Core Layers

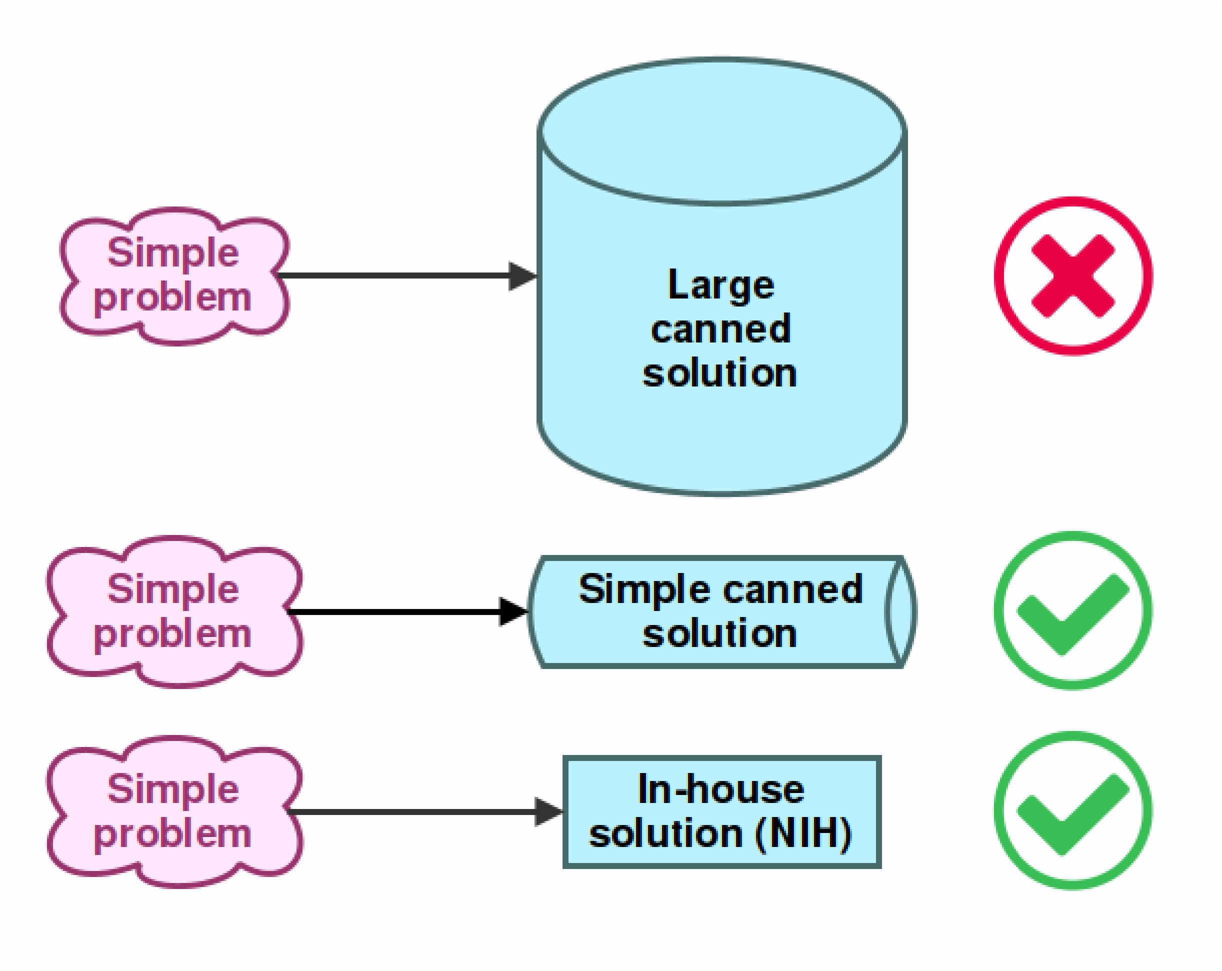
A common reason for wanting to do this is that it's usually less code to simply add a hack in the core layers.

Doing so is not advised. Generally, the code for a solution should be closer to where the problem originates, even if it involves additional, duplicated, more complex, or less efficient code. More creativity might be needed, but this path is always the advised one.

#8: Don't use complex canned solutions for simple problems

Not every problem has a simple solution and, many times, the right choice is to use a third party library to solve the problem.

As Godot requires to be shipped in a large amount of platforms, we can't link libraries dynamically. Instead, we bundle them in our source tree.


Simple problem
Large canned solution
Simple problem
Simple canned solution
Simple problem
In-house solution (NIH)

As a result, we are very picky with what goes in, and we tend to prefer smaller libraries (single header ones are our favorite). We will only bundle something larger if there is no other choice.

Libraries must use a permissive enough license to be included into Godot. Some examples of acceptable licenses are Apache 2.0, BSD, MIT, ISC, and MPL 2.0. In particular, we cannot accept libraries licensed under the GPL or LGPL since these licenses effectively disallow static linking in proprietary software (which Godot is distributed as in most exported projects). This requirement also applies to the editor, since we may want to run it on iOS in the long term. Since iOS doesn't support dynamic linking, static linking is the only option on that platform.

Code style guidelines

When contributing to Godot's source code, you will be expected to follow the style guidelines outlined below. Some of them are checked via the Continuous Integration process and reviewers will ask you to fix potential issues, so best setup your system as outlined below to ensure all your commits follow the guidelines.

C++ and Objective-C

There are no written guidelines, but the code style agreed upon by the developers is enforced via the clang-format [https://clang.llvm.org/docs/ClangFormat.html] code beautifier, which takes care for you of all our conventions. To name a few:

- Indentation and alignment are both tab based (respectively one and two tabs)
- One space around math and assignments operators as well as after commas
- Pointer and reference operators are affixed to the variable identifier, not to the type name
- See further down regarding header includes

The rules used by clang-format are outlined in the .clang-format [https://github.com/godotengine/godot/blob/master/.clang-format] file of the Godot repository.

As long as you ensure that your style matches the surrounding code and that you're not introducing trailing whitespace or space-based indentation, you should be fine. If you plan to contribute regularly, however, we strongly advise that you set up clang-format locally to check and automatically fix all your commits.

Warning

Godot's code style should *not* be applied to third-party code, i.e. that is included in Godot's source tree but was not written specifically for our project. Such code usually comes from different upstream projects with their own style guides (or lack thereof), and don't want to introduce differences that would make syncing with upstream repositories harder.

Third-party code is usually included in the `thirdparty/` folder and can thus easily be excluded from formatting scripts. For the rare cases where a third-party code snippet needs to be included directly within a Godot file, you can use `/* clang-format off */` and `/* clang-format on */` to tell clang-format to ignore a chunk of code.

See also

These guidelines only cover code formatting. See C++ usage guidelines for a list of language features that are permitted in pull requests.

Using clang-format locally

You need to use **clang-format 17** to be compatible with Godot's format. Later versions might be suitable, but previous versions may not support all used options, or format some things differently, leading to style issues in pull requests.

Pre-commit hook

For ease of use, we provide hooks for Git with the pre-commit [https://pre-commit.com/] Python framework that will run clang-format automatically on all your commits with the correct version of clang-format. To set up:

```
pip install pre-commit
pre-commit install
```

You can also run the hook manually with `pre-commit run`.

Note

Previously, we supplied a hook in the folder `misc/hooks`. If you copied the script manually, these hooks should still work, but symlinks will be broken. If you are using the new system, run `rm .git/hooks/*` to remove the old hooks that are no longer needed.

Installation

Here's how to install clang-format:

- Linux: It will usually be available out-of-the-box with the clang toolchain packaged by your distribution. If your distro version is not the required one, you can download a pre-compiled version from the LLVM website [https://releases.llvm.org/download.html], or if you are on a Debian derivative, use the upstream repos [https://apt.llvm.org/].
- macOS and Windows: You can download precompiled binaries from the LLVM website [https://releases.llvm.org/download.html]. You may need to add the path to the binary's folder to your system's `PATH` environment variable to be able to call clang-format out of the box.

You then have different possibilities to apply clang-format to your changes:

Manual usage

You can apply clang-format manually for one or more files with the following command:

```
clang-format -i <path/to/file(s)>
```

- `-i` means that the changes should be written directly to the file (by default clang-format would only output the fixed version to the terminal).
- The path can point to several files, either one after the other or using wildcards like in a typical Unix shell. Be careful when globbing so that you don't run clang-format on compiled objects (.o and .a files) that are in Godot's tree. So better use `core/*.{cpp,h}` than `core/*`.

IDE plugin

Most IDEs or code editors have beautifier plugins that can be configured to run clang-format automatically, for example, each time you save a file.

Here is a non-exhaustive list of beautifier plugins for some IDEs:

- Qt Creator: Beautifier plugin [https://doc.qt.io/qtcreator/creator-beautifier.html]
- Visual Studio Code: Clang-Format [https://marketplace.visualstudio.com/items?itemName=xaver.clang-format]
- Visual Studio: Clang Power Tools 2022 [https://marketplace.visualstudio.com/items?itemName=caphyon.ClangPowerTools2022]
- vim: vim-clang-format [https://github.com/rhysd/vim-clang-format]
- CLion: Starting from version `2019.1`, no plugin is required. Instead, enable ClangFormat [https://www.jetbrains.com/help/clion/clangformat-as-alternative-formatter.html#clion-support]

(Pull requests are welcome to extend this list with tested plugins.)

Header includes

When adding new C++ or Objective-C files or including new headers in existing ones, the following rules should be followed:

- The first lines in the file should be Godot's copyright header and MIT license, copy-pasted from another file. Make sure to adjust the filename.
- In a `.h` header, include guards should be used with the form `FILENAME_H`.
- In a `.cpp` file (e.g. `filename.cpp`), the first include should be the one where the class is declared (e.g. `#include "filename.h"`), followed by an empty line for separation.
- Then come headers from Godot's own code base, included in alphabetical order (enforced by `clang-format`) with paths relative to the root folder. Those includes should be done with quotes, e.g. `#include "core/object.h"`. The block of Godot header includes should then be followed by an empty line for separation.
- Finally, third-party headers (either from `thirdparty` or from the system's include paths) come next and should be included with the < and > symbols, e.g. `#include <png.h>`. The block of third-party headers should also be followed by an empty line for separation.
- Godot and third-party headers should be included in the file that requires them, i.e. in the *.h* header if used in the declarative code or in the *.cpp* if used only in the imperative code.

Example:

```
/*******************************************************************
************/
/*  my_new_file.h
*/
/*******************************************************************
************/
```

```
/*                          This file is part of:
*/
/*                              GODOT ENGINE
*/
/*                         https://godotengine.org
*/
/**************************************************************
************/
/* Copyright (c) 2014-present Godot Engine contributors (see
AUTHORS.md). */
/* Copyright (c) 2007-2014 Juan Linietsky, Ariel Manzur.
*/
/*
*/
/* Permission is hereby granted, free of charge, to any person
obtaining  */
/* a copy of this software and associated documentation files
(the        */
/* "Software"), to deal in the Software without restriction,
including    */
/* without limitation the rights to use, copy, modify, merge,
publish,    */
/* distribute, sublicense, and/or sell copies of the Software,
and to      */
/* permit persons to whom the Software is furnished to do so,
subject to  */
/* the following conditions:
*/
/*
*/
/* The above copyright notice and this permission notice shall
be          */
/* included in all copies or substantial portions of the
Software.        */
/*
*/
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND,         */
/* EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF     */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. */
/* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
```

```
FOR ANY    */
/* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,    */
/* TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE       */
/* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
/*************************************************************
************/

#ifndef MY_NEW_FILE_H
#define MY_NEW_FILE_H

#include "core/hash_map.h"
#include "core/list.h"
#include "scene/gui/control.h"

#include <png.h>

...

#endif // MY_NEW_FILE_H
```

```
/*************************************************************
************/
/*  my_new_file.cpp
*/
/*************************************************************
************/
/*                           This file is part of:
*/
/*                               GODOT ENGINE
*/
/*                          https://godotengine.org
*/
/*************************************************************
************/
/* Copyright (c) 2014-present Godot Engine contributors (see
AUTHORS.md). */
/* Copyright (c) 2007-2014 Juan Linietsky, Ariel Manzur.
*/
/*
```

```
*/
/* Permission is hereby granted, free of charge, to any person
obtaining  */
/* a copy of this software and associated documentation files
(the        */
/* "Software"), to deal in the Software without restriction,
including    */
/* without limitation the rights to use, copy, modify, merge,
publish,    */
/* distribute, sublicense, and/or sell copies of the Software,
and to     */
/* permit persons to whom the Software is furnished to do so,
subject to  */
/* the following conditions:
*/
/*
*/
/* The above copyright notice and this permission notice shall
be         */
/* included in all copies or substantial portions of the
Software.        */
/*
*/
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND,        */
/* EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF     */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. */
/* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY   */
/* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,   */
/* TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE      */
/* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
/**************************************************************
************/

#include "my_new_file.h"

#include "core/math/math_funcs.h"
```

```
#include "scene/gui/line_edit.h"

#include <zlib.h>
#include <zstd.h>
```

Java

Godot's Java code (mostly in `platform/android`) is also enforced via `clang-format`, so see the instructions above to set it up. Keep in mind that this style guide only applies to code written and maintained by Godot, not third-party code such as the `java/src/com/google` subfolder.

Python

Godot's SCons buildsystem is written in Python, and various scripts included in the source tree are also using Python.

For those, we use the Ruff linter and code formatter [https://docs.astral.sh/ruff/].

Using ruff locally

First of all, you will need to install Ruff. Ruff requires Python 3.7+ to run.

Installation

Here's how to install ruff:

```
pip3 install ruff --user
```

You then have different possibilities to apply ruff to your changes:

Manual usage

You can apply `ruff` manually to one or more files with the following command:

```
ruff -l 120 <path/to/file(s)>
```

- `-l 120` means that the allowed number of characters per line is 120. This number was agreed upon by the developers.
- The path can point to several files, either one after the other or using wildcards like in a typical Unix shell.

Pre-commit hook

For ease of use, we provide hooks for Git with the pre-commit [https://pre-commit.com/] Python framework that will run `ruff` automatically on all your commits with the correct version of `ruff`. To set up:

```
pip install pre-commit
pre-commit install
```

You can also run the hook manually with `pre-commit run`.

Note

Previously, we supplied a hook in the folder `misc/hooks`. If you copied the script manually, these hooks should still work, but symlinks will be broken. If you are using the new system, run `rm .git/hooks/*` to remove the old hooks that are no longer needed.

Editor integration

Many IDEs or code editors have beautifier plugins that can be configured to run ruff automatically, for example, each time you save a file. For details, you can check Ruff Integrations [https://docs.astral.sh/ruff/integrations/].

Comment style guide

This comment style guide applies to all programming languages used within Godot's codebase.

- Begin comments with a space character to distinguish them from disabled code.
- Use sentence case for comments. Begin comments with an uppercase character and always end them with a period.
- Reference variable/function names and values using backticks.
- Wrap comments to ~100 characters.
- You can use `TODO:`, `FIXME:`, `NOTE:`, or `HACK:` as admonitions when needed.

Example:

```
// Compute the first 10,000 decimals of Pi.
// FIXME: Don't crash when computing the 1,337th decimal due to `increment`
//        being negative.
```

Don't repeat what the code says in a comment. Explain the *why* rather than *how*.

Bad:

```
// Draw loading screen.
draw_load_screen();
```

You can use Javadoc-style comments above function or macro definitions. It's recommended to use Javadoc-style comments *only* for methods which are not exposed to scripting. This is because exposed methods should be documented in the class reference XML instead.

Example:

```
/**
 * Returns the number of nodes in the universe.
 * This can potentially be a very large number, hence the 64-bit return type.
 */
uint64_t Universe::get_node_count() {
    // ...
}
```

For member variables, don't use Javadoc-style comments but use single-line comments instead:

```
class Universe {
    // The cached number of nodes in the universe.
    // This value may not always be up-to-date with the current number of nodes
    // in the universe.
    uint64_t node_count_cached = 0;
};
```

C++ usage guidelines

Rationale

Since Godot 4.0, the C++ standard used throughout the codebase is a subset of **C++17**. While modern C++ brings a lot of opportunities to write faster, more readable code, we chose to restrict our usage of C++ to a subset for a few reasons:

- It makes it easier to review code in online editors. This is because engine contributors don't always have access to a full-featured IDE while reviewing code.
- It makes the code easier to grasp for beginner contributors (who may not be professional C++ programmers). Godot's codebase is known to be easy to learn from, and we'd like to keep it that way.

To get your pull request merged, it needs to follow the C++ usage guidelines outlined here. Of course, you can use features not allowed here in your own C++ modules or GDExtensions.

Note

Prior to Godot 4.0, the C++ standard used throughout the codebase was C++03, with a handful of C++14 extensions. If you are contributing a pull request to the *3.x* branch rather than *master*, your code can't use C++17 features. Instead, your code must be able to be built with a C++14 compiler.

The guidelines below don't apply to third-party dependencies, although we generally favor small libraries instead of larger solutions. See also Best practices for engine contributors.

See also

See Code style guidelines for formatting guidelines.

Disallowed features

Any feature not listed below is allowed. Using features like `constexpr` variables and `nullptr` is encouraged when possible. Still, try to keep your use of modern C++ features conservative. Their use needs to serve a real purpose, such as improving code readability or performance.

Standard Template Library

We don't allow using the STL [https://en.wikipedia.org/wiki/Standard_Template_Library] as Godot provides its own data types (among other things). See Why does Godot not use STL (Standard Template Library)? for more information.

This means that pull requests should **not** use `std::string`, `std::vector` and the like. Instead, use Godot's datatypes as described below:

- Use `String` instead of `std::string`.
- Use `Vector` instead of `std::vector`. In some cases, `LocalVector` can be used as an alternative (ask core developers first).
- Use `Array` instead of `std::array`.

Note

Godot also has a List datatype (which is a linked list). While List is already used in the codebase, it typically performs worse than other datatypes like Vector and Array. Therefore, List should be avoided in new code unless necessary.

`auto` keyword

Please don't use the `auto` keyword for type inference. While it can avoid repetition, it can also lead to confusing code:

```
// Not so confusing...
auto button = memnew(Button);

// ...but what about this?
auto result = EditorNode::get_singleton()-
>get_complex_result();
```

Keep in mind hover documentation often isn't readily available for pull request reviewers. Most of the time, reviewers will use GitHub's online viewer to review pull requests.

We chose to forbid `auto` instead of allowing it on a case-by-case basis to avoid having to decide on difficult edge cases. Thank you for your understanding.

Lambdas

Lambdas should be used conservatively when they make code effectively faster or simpler, and do not impede readability. Please ask before using lambdas in a pull request.

`#pragma once` directive

To follow the existing style, please use standard `#ifdef`-based include guards instead of `#pragma once` in new files.

See also

See Header includes for guidelines on sorting includes in C++ and Objective-C files.

Handling compatibility breakages

So you've added a new parameter to a method, changed the return type, changed the type of a parameter, or changed its default value, and now the automated testing is complaining about compatibility breakages?

Breaking compatibility should be avoided, but when necessary there are systems in place to handle this in a way that makes the transition as smooth as possible.

A practical example

These changes are taken from pull request #88047 [https://github.com/godotengine/godot/pull/88047], which added new pathing options to `AStarGrid2D` and other AStar classes. Among other changes, these methods were modified in `core/math/a_star_grid_2d.h`:

```
Vector<Vector2> get_point_path(const Vector2i &p_from, const Vector2i &p_to);
TypedArray<Vector2i> get_id_path(const Vector2i &p_from, const Vector2i &p_to);
```

To:

```
Vector<Vector2> get_point_path(const Vector2i &p_from, const Vector2i &p_to, bool p_allow_partial_path = false);
TypedArray<Vector2i> get_id_path(const Vector2i &p_from, const Vector2i &p_to, bool p_allow_partial_path = false);
```

This meant adding new compatibility method bindings to the file, which should be in the `protected` section of the code, usually placed next to `_bind_methods()`:

```
#ifndef DISABLE_DEPRECATED
    TypedArray<Vector2i> _get_id_path_bind_compat_88047(const
Vector2i &p_from, const Vector2i &p_to);
    Vector<Vector2> _get_point_path_bind_compat_88047(const
Vector2i &p_from, const Vector2i &p_to);
    static void _bind_compatibility_methods();
#endif
```

They should start with a _ to indicate that they are internal, and end with `_bind_compat_` followed by the PR number that introduced the change (`88047` in this example). These compatibility methods need to be implemented in a dedicated file, like `core/math/a_star_grid_2d.compat.inc` in this case:

```
/***************************************************************
***********/
/*  a_star_grid_2d.compat.inc
*/
/***************************************************************
***********/
/*                         This file is part of:
*/
/*                             GODOT ENGINE
*/
/*                        https://godotengine.org
*/
/***************************************************************
***********/
/* Copyright (c) 2014-present Godot Engine contributors (see
AUTHORS.md). */
/* Copyright (c) 2007-2014 Juan Linietsky, Ariel Manzur.
*/
/*
*/
/* Permission is hereby granted, free of charge, to any person
obtaining  */
/* a copy of this software and associated documentation files
```

```
(the         */
/* "Software"), to deal in the Software without restriction,
including    */
/* without limitation the rights to use, copy, modify, merge,
publish,    */
/* distribute, sublicense, and/or sell copies of the Software,
and to      */
/* permit persons to whom the Software is furnished to do so,
subject to  */
/* the following conditions:
*/
/*
*/
/* The above copyright notice and this permission notice shall
be          */
/* included in all copies or substantial portions of the
Software.        */
/*
*/
/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND,        */
/* EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF     */
/* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. */
/* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY    */
/* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,   */
/* TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE      */
/* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
/**************************************************************
************/

#ifndef DISABLE_DEPRECATED

#include "core/variant/typed_array.h"

TypedArray<Vector2i>
AStarGrid2D::_get_id_path_bind_compat_88047(const Vector2i
&p_from_id, const Vector2i &p_to_id) {
    return get_id_path(p_from_id, p_to_id, false);
}

Vector<Vector2>
AStarGrid2D::_get_point_path_bind_compat_88047(const Vector2i
&p_from_id, const Vector2i &p_to_id) {
    return get_point_path(p_from_id, p_to_id, false);
}

void AStarGrid2D::_bind_compatibility_methods() {
    ClassDB::bind_compatibility_method(D_METHOD("get_id_path",
"from_id", "to_id"),
&AStarGrid2D::_get_id_path_bind_compat_88047);

ClassDB::bind_compatibility_method(D_METHOD("get_point_path",
"from_id", "to_id"),
&AStarGrid2D::_get_point_path_bind_compat_88047);
}

#endif // DISABLE_DEPRECATED
```

Unless the change in compatibility is complex, the compatibility method should simply call the modified method directly, instead of duplicating that method. Make sure to match the default arguments for that method (in the example above this would be `false`).

This file should always be placed next to the original file, and have `.compat.inc` at the end instead of `.cpp` or `.h`. Next, this should be included in the `.cpp` file we're adding compatibility methods to, so `core/math/a_star_grid_2d.cpp`:

```
#include "a_star_grid_2d.h"
#include "a_star_grid_2d.compat.inc"

#include "core/variant/typed_array.h"
```

And finally, the changes reported by the API validation step should be added to the relevant validation file. Because this was done during the development of 4.3, this would be `misc/extension_api_validation/4.2-stable.expected` (including changes not shown in this example):

```
GH-88047
--------
Validate extension JSON: Error: Field
'classes/AStar2D/methods/get_id_path/arguments': size changed
value in new API, from 2 to 3.
Validate extension JSON: Error: Field
'classes/AStar2D/methods/get_point_path/arguments': size
changed value in new API, from 2 to 3.
Validate extension JSON: Error: Field
'classes/AStar3D/methods/get_id_path/arguments': size changed
value in new API, from 2 to 3.
Validate extension JSON: Error: Field
'classes/AStar3D/methods/get_point_path/arguments': size
changed value in new API, from 2 to 3.
Validate extension JSON: Error: Field
'classes/AStarGrid2D/methods/get_id_path/arguments': size
changed value in new API, from 2 to 3.
Validate extension JSON: Error: Field
'classes/AStarGrid2D/methods/get_point_path/arguments': size
changed value in new API, from 2 to 3.

Added optional "allow_partial_path" argument to get_id_path and
get_point_path methods in AStar classes.
Compatibility methods registered.
```

The instructions for how to add to that file are at the top of the file itself.

If you get a "Hash changed" error for a method, it means that the compatibility binding is missing or incorrect. Such lines shouldn't be added to the `.expected` file, but fixed by binding the proper compatibility method.

And that's it! You might run into a bit more complicated cases, like rearranging arguments, changing return types, etc., but this covers the basic on how to use this system.

For more information, see pull request #76446 [https://github.com/godotengine/godot/pull/76446].

Configuring an IDE

We assume that you have already cloned [https://github.com/godotengine/godot] and compiled Godot.

You can easily develop Godot with any text editor and by invoking `scons` on the command line, but if you want to work with an IDE (Integrated Development Environment), here are setup instructions for some popular ones:

- Android Studio
- CLion
- Code::Blocks
- KDevelop
- Qt Creator
- Rider
- Visual Studio
- Visual Studio Code
- Xcode

It is possible to use other IDEs, but their setup is not documented yet.

If your editor supports the language server protocol [https://microsoft.github.io/language-server-protocol/], you can use clangd [https://clangd.llvm.org] for completion, diagnostics, and more. You can generate a compilation database for use with clangd one of two ways:

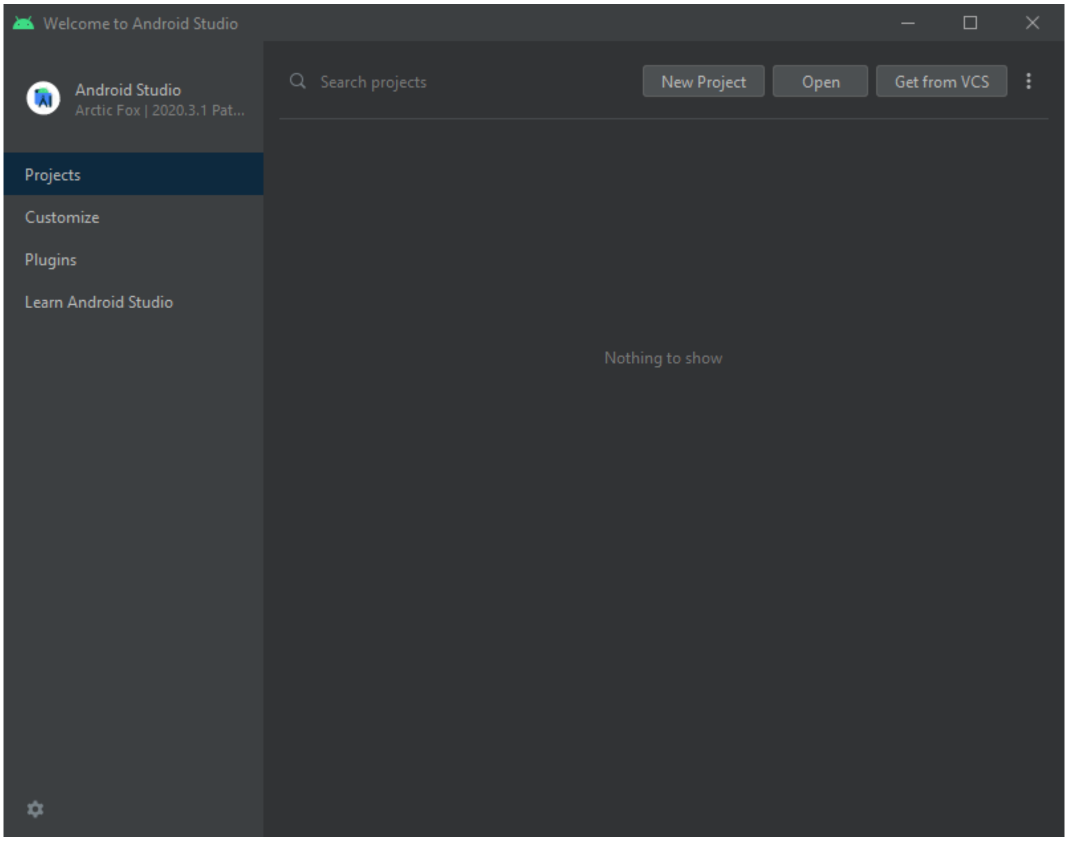
```
# Generate compile_commands.json while compiling
scons compiledb=yes

# Generate compile_commands.json without compiling
scons compiledb=yes compile_commands.json
```

Android Studio

Android Studio [https://developer.android.com/studio] is a free IDE for Android development made by Google [https://about.google/] and JetBrains [https://www.jetbrains.com/]. It's based on IntelliJ IDEA [https://www.jetbrains.com/idea/] and has a feature-rich editor which supports Java and C/C++. It can be used to work on Godot's core engine as well as the Android platform codebase.

Importing the project

- From the Android Studio's welcome window select **Open**.


Welcome to Android Studio
Android Studio
Arctic Fox | 2020.3.1 Pat...
Search projects
New Project
Open
Get from VCS
Projects
Customize
Plugins
Learn Android Studio
Nothing to show

Android Studio's welcome window.

- Navigate to `<Godot root directory>/platform/android/java` and select the `settings.gradle` file.
- Android Studio will import and index the project.

Android Studio project layout

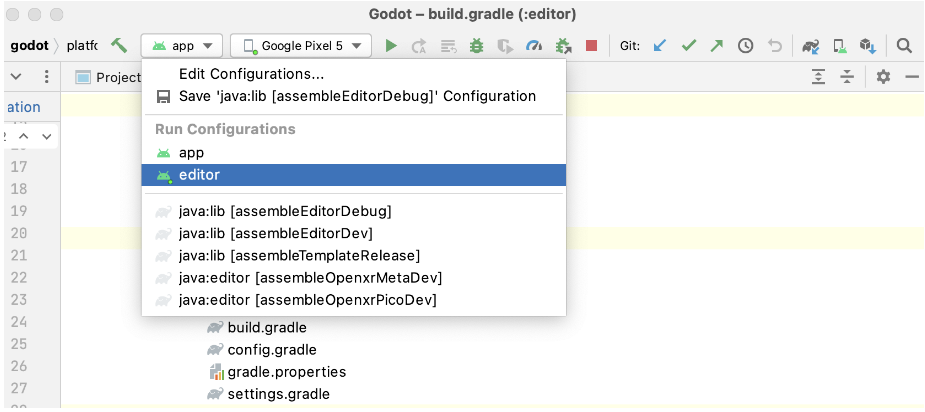
The project is organized using Android Studio's modules [https://developer.android.com/studio/projects#ApplicationModules]:

- `lib` module:
    - Located under `<Godot root directory>/platform/android/java/lib`, this is a **library module** that organizes the Godot java and native code and make it available as a reusable dependency / artifact.
    - The artifact generated by this module is made available for other Android modules / projects to use as a dependency, via MavenCentral [https://repo1.maven.org/maven2/org/godotengine/godot/].
- `editor` module:
    - Located under `<Godot root directory>/platform/android/java/editor`, this is an **application module** that holds the source code for the Android port of the Godot Editor.
    - This module has a dependency on the `lib` module.
- `app` module:
    - Located under `<Godot root directory>/platform/android/java/app`, this is an **application module** that holds the source code for the Android build templates.

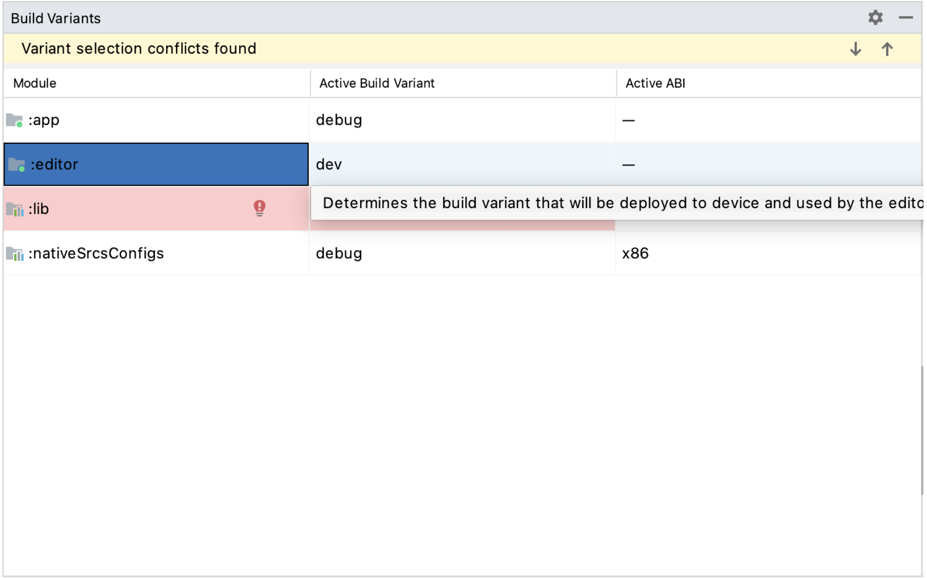
  - This module has a dependency on the `lib` module.

Building & debugging the editor module

- To build the `editor` module:
  - Select the [Run/Debug Configurations drop down](https://developer.android.com/studio/run/rundebugconfig#running) [https://developer.android.com/studio/run/rundebugconfig#running] and select `editor`.


Godot – build.gradle (:editor)
godot
platfc
app
Google Pixel 5
Git:
Project
ation
Edit Configurations...
Save 'java:lib [assembleEditorDebug]' Configuration
Run Configurations
app
editor
java:lib [assembleEditorDebug]
java:lib [assembleEditorDev]
java:lib [assembleTemplateRelease]
java:editor [assembleOpenxrMetaDev]
java:editor [assembleOpenxrPicoDev]
build.gradle
config.gradle
gradle.properties
settings.gradle
17
18
19
20
21
22
23
24
25
26
27

  - Select **Run > Run 'editor'** from the top menu or [click the Run icon](https://developer.android.com/studio/run/rundebugconfig#running) [https://developer.android.com/studio/run/rundebugconfig#running].
- To debug the `editor` module:
  - Open the **Build Variants** window using **View > Tools Windows > Build Variants** from the top menu.
  - In the **Build Variants** window, make sure that in the **Active Build Variant** column, the `:editor` entry is set to **dev**.

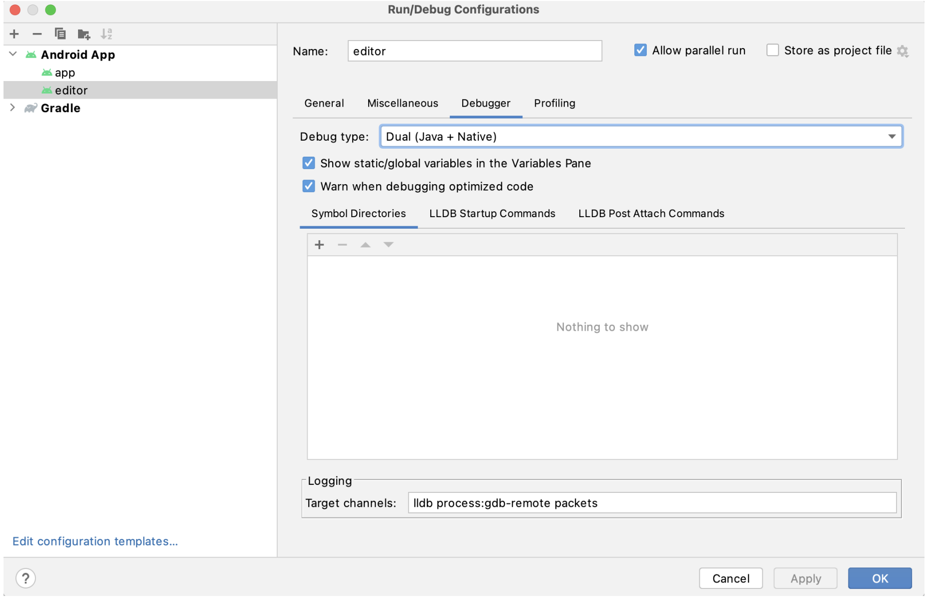

Build Variants

Variant selection conflicts found

Module	Active Build Variant	Active ABI
:app	debug	—
:editor	dev	—
:lib		
:nativeSrcsConfigs	debug	x86

Determines the build variant that will be deployed to device and used by the edito

- Open the **Run/Debug Configurations** window by clicking on **Run > Edit Configurations...** on the top menu.
- In the **Run/Debug Configurations** window, select the `editor` entry, and under **Debugger** make sure the **Debug Type** is set to `Dual (Java + Native)`


Run/Debug Configurations
Android App
app
editor
Gradle
Edit configuration templates...
General
Miscellaneous
Debugger
Profiling
Symbol Directories
LLDB Startup Commands
LLDB Post Attach Commands
Nothing to show
Cancel
Apply
OK

Name: editor ☑ Allow parallel run ☐ Store as project file

Debug type: Dual (Java + Native)

☑ Show static/global variables in the Variables Pane

☑ Warn when debugging optimized code

Logging

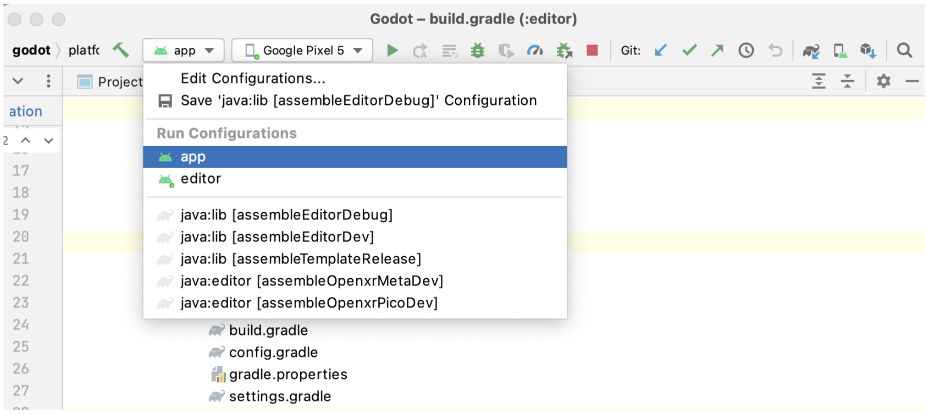
Target channels: lldb process:gdb-remote packets

- Select **Run > Debug 'editor'** from the top menu or click the Debug icon [https://developer.android.com/studio/run/rundebugconfig#running].

Building & debugging the app module

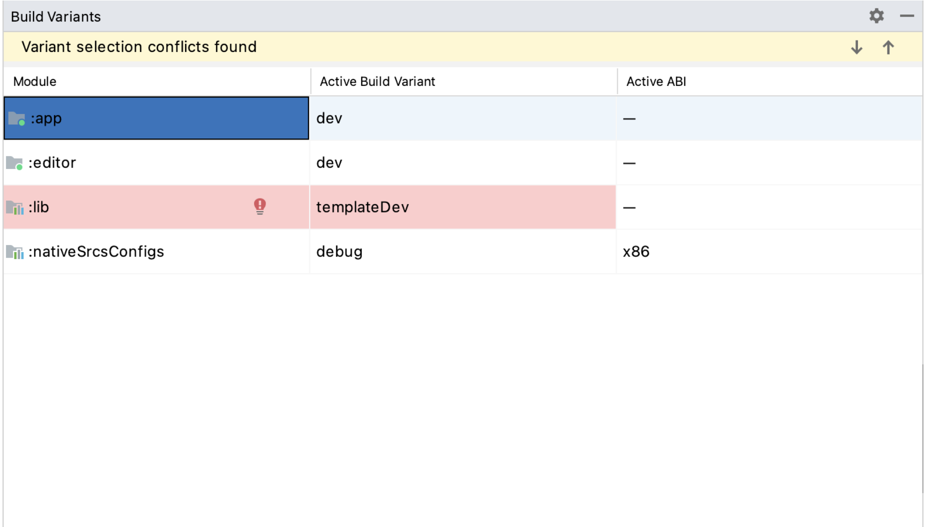
The `app` module requires the presence of a Godot project in its `assets` directory (`<Godot root directory>/platform/android/java/app/assets`) to run. This is usually handled by the Godot Editor during the export process. While developing in Android Studio, it's necessary to manually add a Godot project under that directory to replicate the export process. Once that's done, you can follow the instructions below to run/debug the `app` module:

- To build the `app` module:
    - Select the Run/Debug Configurations drop down [https://developer.android.com/studio/run/rundebugconfig#running] and select `app`.


Godot – build.gradle (:editor)
godot
platfc
app
Google Pixel 5
Git:
Project
ation
Edit Configurations...
Save 'java:lib [assembleEditorDebug]' Configuration
Run Configurations
app
editor
java:lib [assembleEditorDebug]
java:lib [assembleEditorDev]
java:lib [assembleTemplateRelease]
java:editor [assembleOpenxrMetaDev]
java:editor [assembleOpenxrPicoDev]
build.gradle
config.gradle
gradle.properties
settings.gradle

    - Select **Run > Run 'app'** from the top menu or click the Run icon [https://developer.android.com/studio/run/rundebugconfig#running].

- To debug the `app` module:
    - Open the **Build Variants** window using **View > Tools Windows > Build Variants** from the top menu.
    - In the **Build Variants** window, make sure that in the **Active Build Variant** column, the `:app` entry is set to **dev**.

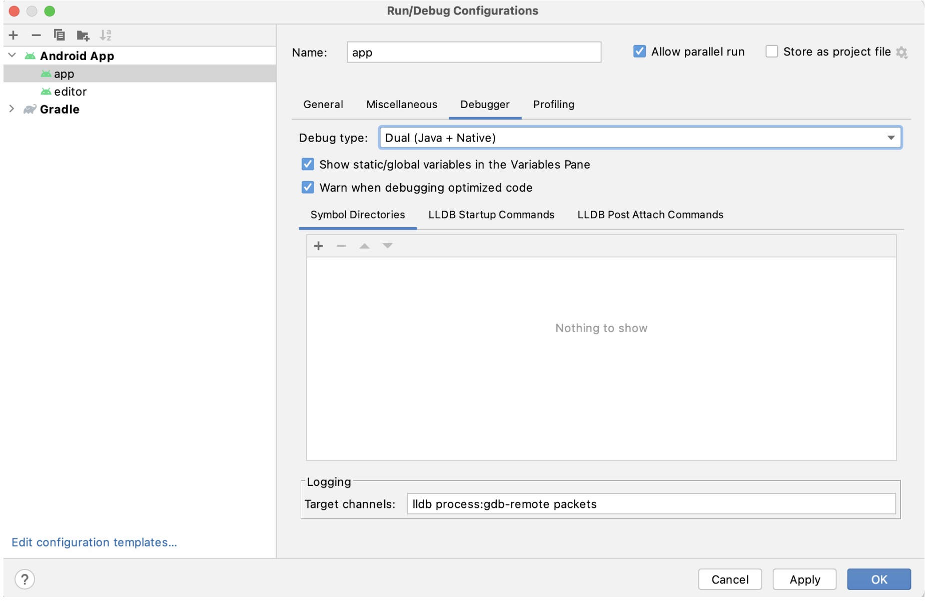


Build Variants

Variant selection conflicts found

Module	Active Build Variant	Active ABI
:app	dev	—
:editor	dev	—
:lib	templateDev	—
:nativeSrcsConfigs	debug	x86

    - Open the **Run/Debug Configurations** window by clicking on **Run > Edit Configurations...** on the top menu.
    - In the **Run/Debug Configurations** window, select the `app` entry, and under **Debugger** make sure the **Debug Type** is set to `Dual (Java + Native)`


Run/Debug Configurations
Android App
app
editor
Gradle
General
Miscellaneous
Debugger
Profiling
Symbol Directories
LLDB Startup Commands
LLDB Post Attach Commands
Nothing to show
Logging
Edit configuration templates...
Cancel
Apply
OK

Name: app

☑ Allow parallel run

☐ Store as project file

Debug type: Dual (Java + Native)

☑ Show static/global variables in the Variables Pane

☑ Warn when debugging optimized code

Target channels: lldb process:gdb-remote packets

- Select **Run > Debug 'app'** from the top menu or [click the Debug icon](https://developer.android.com/studio/run/rundebugconfig#running) [https://developer.android.com/studio/run/rundebugconfig#running].

If you run into any issues, ask for help in [Godot's Android dev channel](https://chat.godotengine.org/channel/android) [https://chat.godotengine.org/channel/android].

CLion

CLion [https://www.jetbrains.com/clion/] is a commercial JetBrains [https://www.jetbrains.com/] IDE for C++.

Importing the project

CLion can import a project's compilation database file [https://clang.llvm.org/docs/JSONCompilationDatabase.html], commonly named `compile_commands.json`. To generate the compilation database file, open the terminal, change to the Godot root directory, and run:

```
scons compiledb=yes
```

Then, open the Godot root directory with CLion. CLion will import the compilation database, index the codebase, and provide autocompletion and other advanced code navigation and refactoring functionality.

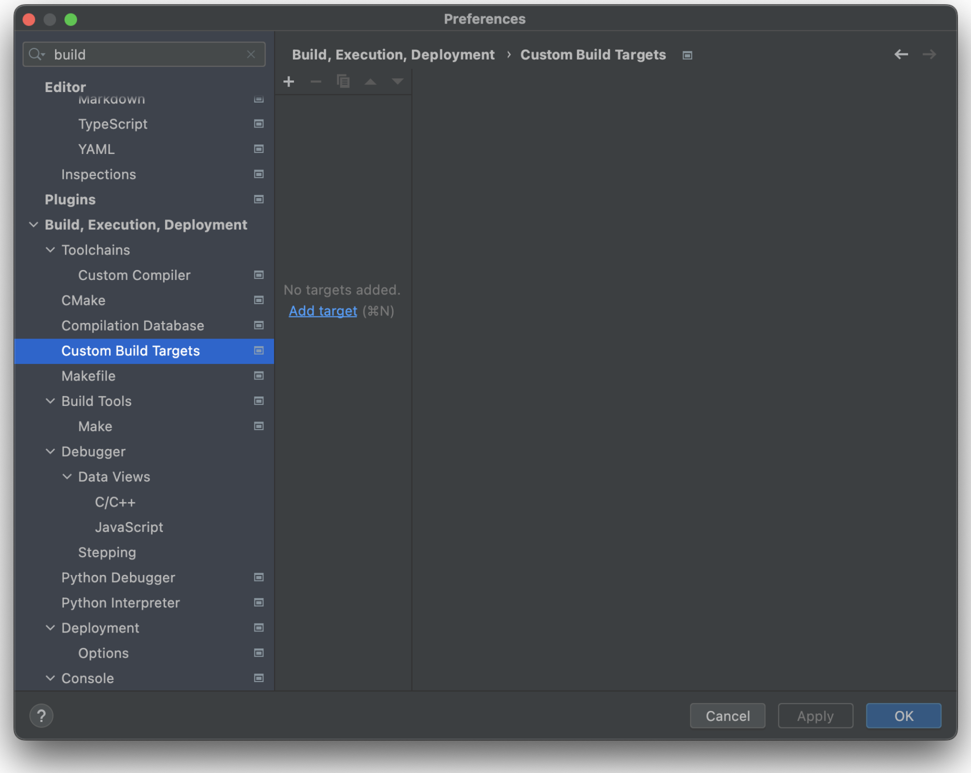
Compiling and debugging the project

CLion does not support compiling and debugging Godot via SCons out of the box. This can be achieved by creating a custom build target and run configuration in CLion. Before creating a custom build target, you must compile Godot once on the command line, to generate the Godot executable. Open the terminal, change into the Godot root directory, and execute:

```
scons dev_build=yes
```

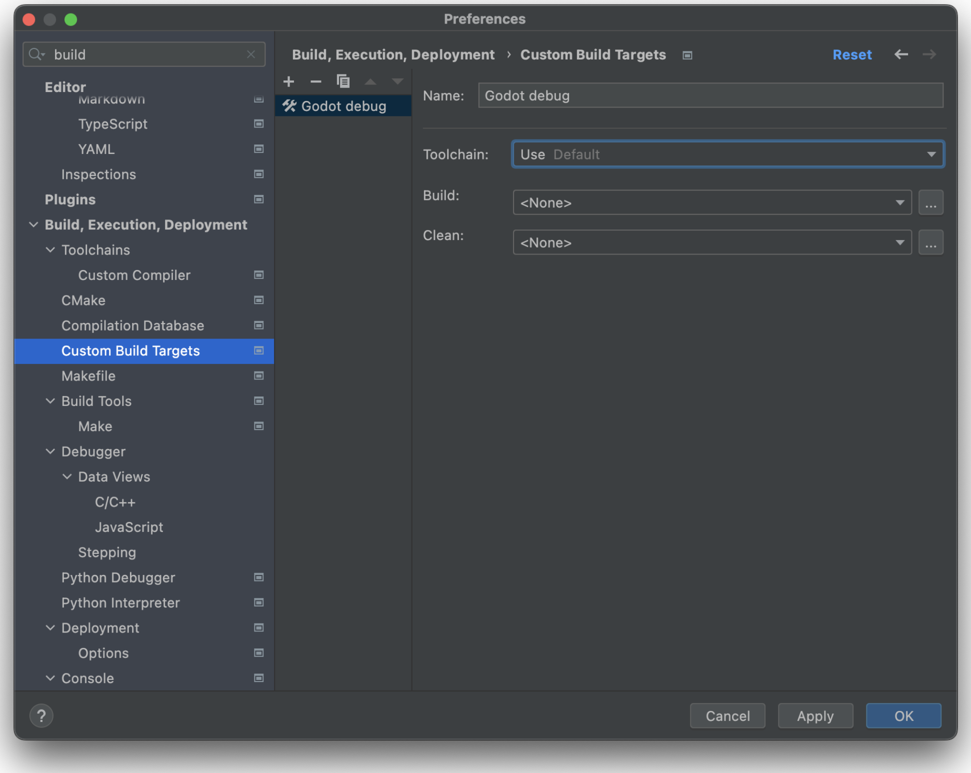
To add a custom build target that invokes SCons for compilation:

- Open CLion and navigate to **Preferences > Build, Execution, Deployment > Custom Build Targets**


Preferences
build
Editor
Markdown
TypeScript
YAML
Inspections
Plugins
Build, Execution, Deployment
Toolchains
Custom Compiler
CMake
Compilation Database
Custom Build Targets
Makefile
Build Tools
Make
Debugger
Data Views
C/C++
JavaScript
Stepping
Python Debugger
Python Interpreter
Deployment
Options
Console
Build, Execution, Deployment › Custom Build Targets
Add target (⌘N)
?
Cancel
Apply
OK

No targets added.

- Click **Add target** and give the target a name, e.g. `Godot debug`.



Preferences
build
Editor
Markdown
TypeScript
YAML
Inspections
Plugins
Build, Execution, Deployment
Toolchains
Custom Compiler
CMake
Compilation Database
Custom Build Targets
Makefile
Build Tools
Make
Debugger
Data Views
C/C++
JavaScript
Stepping
Python Debugger
Python Interpreter
Deployment
Options
Console
Build, Execution, Deployment › Custom Build Targets
Reset
Godot debug
...
...
?
Cancel
Apply
OK

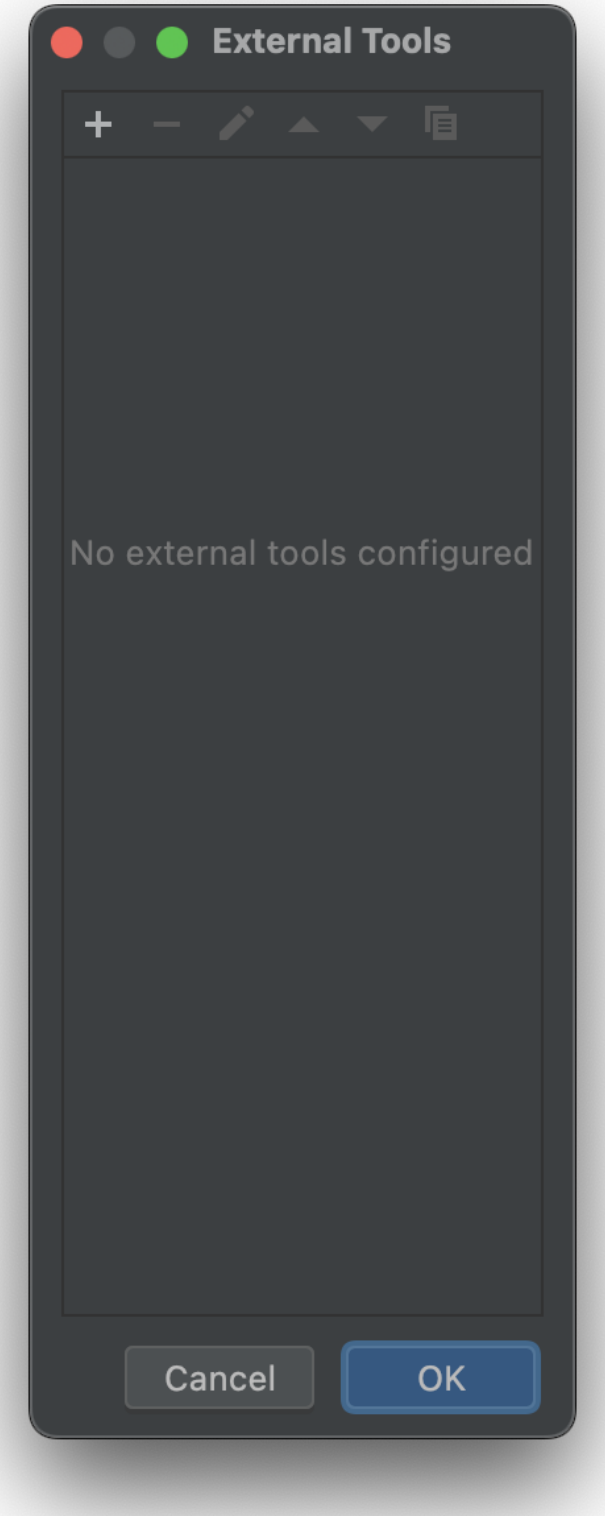
Name: Godot debug

Toolchain: Use Default

Build: <None>

Clean: <None>

- Click **...** next to the **Build:** selectbox, then click the **+** button in the **External Tools** dialog to add a new external tool.

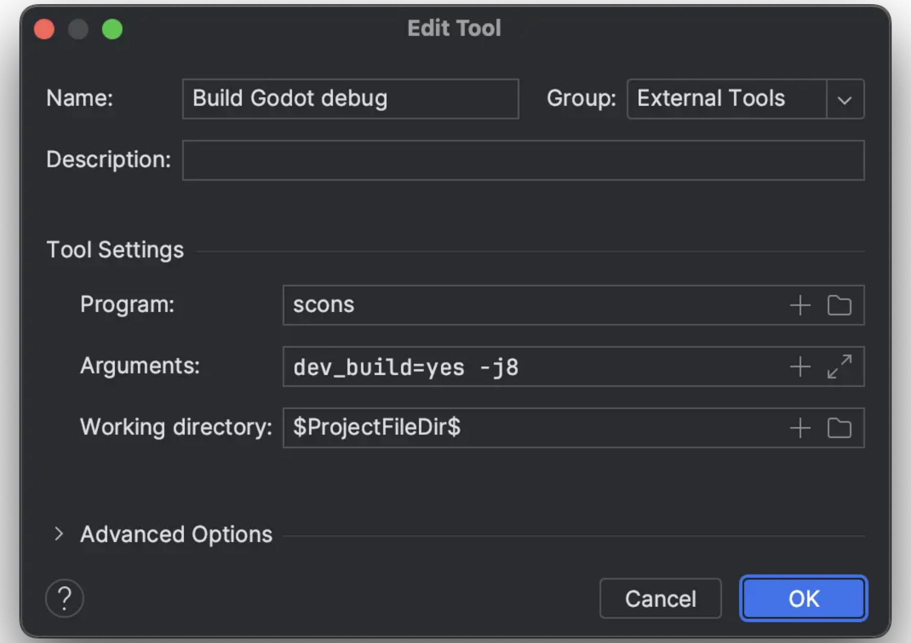

External Tools

No external tools configured

- Give the tool a name, e.g. `Build Godot debug`, set **Program** to `scons`, set **Arguments** to the compilation settings you want (see compiling Godot), and set the **Working directory** to `$ProjectFileDir$`, which equals the Godot root directory. Click **OK** to create the tool.

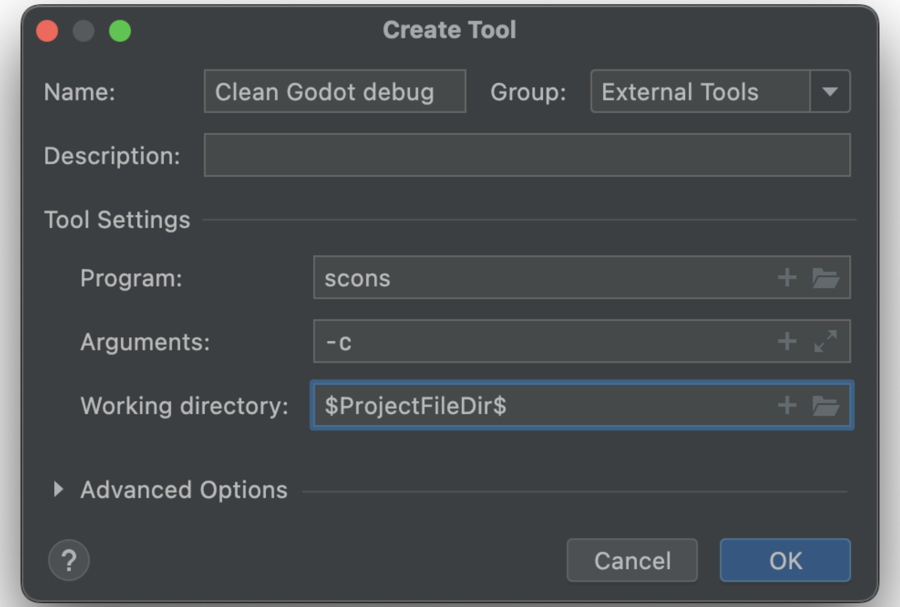
  ### Note

  CLion does not expand shell commands like `scons -j$(nproc)`. Use concrete values instead, e.g. `scons -j8`.



Name: Build Godot debug
Group: External Tools
Description:
Tool Settings
Program: scons
Arguments: dev_build=yes -j8
Working directory: $ProjectFileDir$

- Back in the **External Tools** dialog, click the **+** again to add a second external tool for cleaning the Godot build via SCons. Give the tool a name, e.g. `Clean Godot debug`, set **Program** to `scons`, set **Arguments** to `-c` (which will clean the build), and set the **Working directory** to `$ProjectFileDir$`. Click **OK** to create the tool.



Name: Clean Godot debug

Group: External Tools

Description:

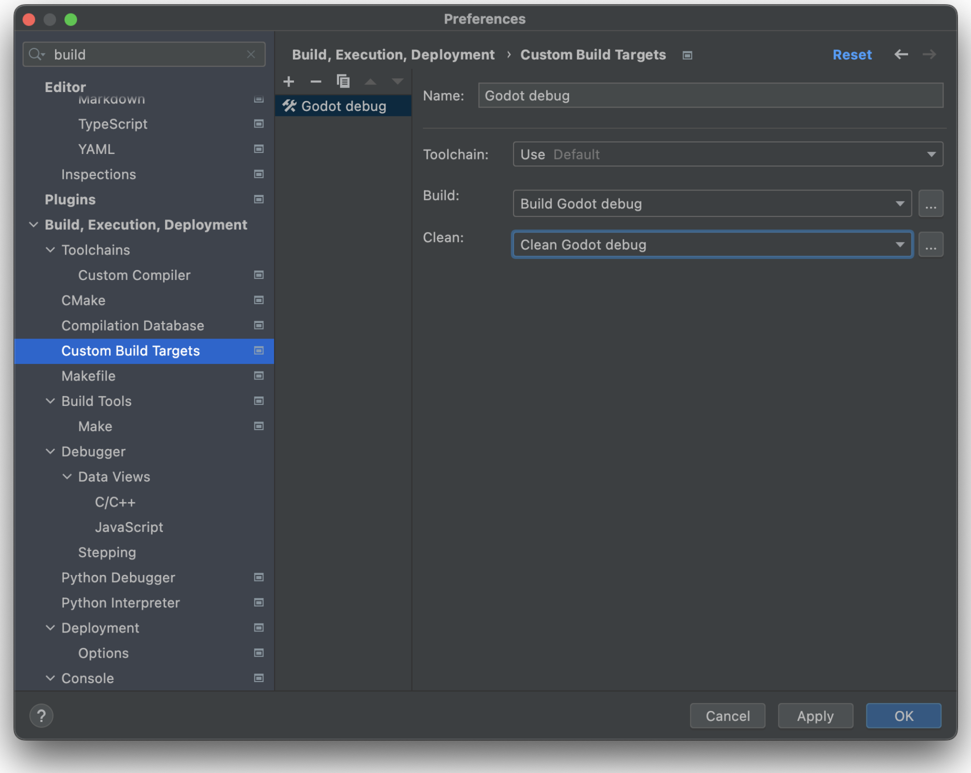
Tool Settings

Program: scons

Arguments: -c

Working directory: $ProjectFileDir$

- Close the **External Tools** dialog. In the **Custom Build Target** dialog for the custom `Godot debug` build target, select the **Build Godot debug** tool from the **Build** select box, and select the **Clean Godot debug** tool from the **Clean** select box. Click **OK** to create the custom build target.


Preferences
build
Editor
Markdown
TypeScript
YAML
Inspections
Plugins
Build, Execution, Deployment
Toolchains
Custom Compiler
CMake
Compilation Database
Custom Build Targets
Makefile
Build Tools
Make
Debugger
Data Views
C/C++
JavaScript
Stepping
Python Debugger
Python Interpreter
Deployment
Options
Console
Build, Execution, Deployment › Custom Build Targets
Reset
Godot debug
Cancel
Apply
OK

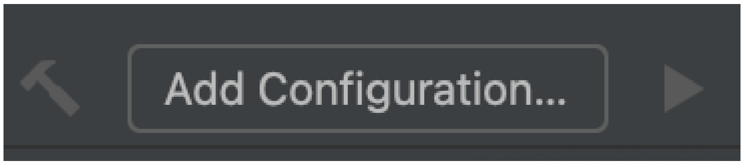
Name: Godot debug

Toolchain: Use Default

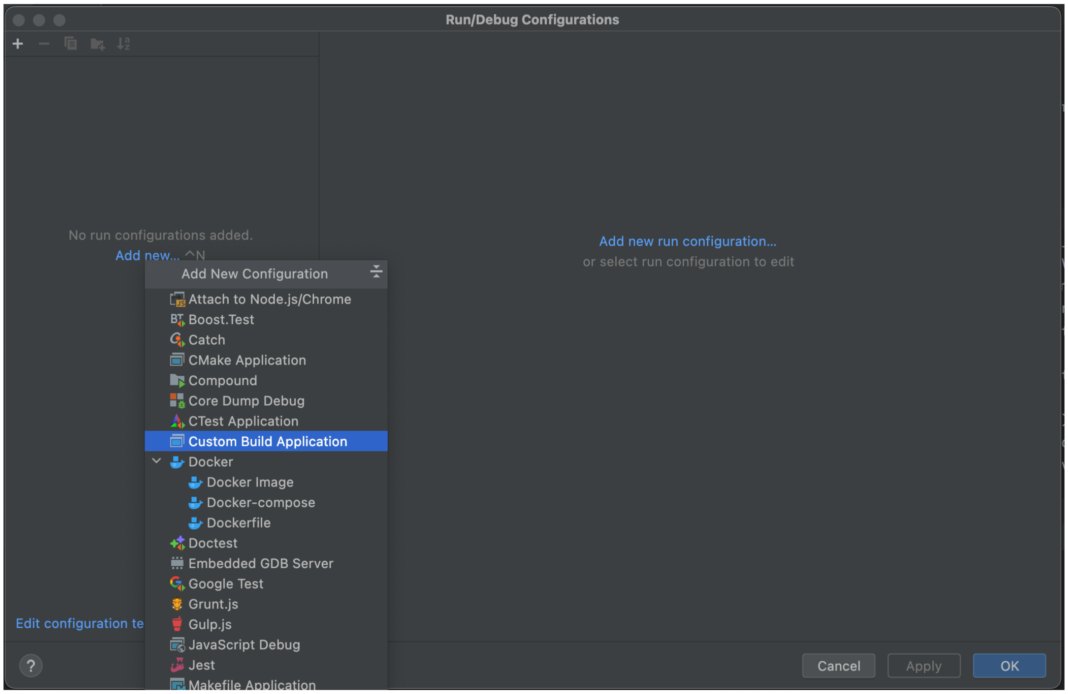
Build: Build Godot debug

Clean: Clean Godot debug

- In the main IDE window, click **Add Configuration**.


Add Configuration...

- In the **Run/Debug Configuration** dialog, click **Add new...**, then select **Custom Build Application** to create a new custom run/debug configuration.



Run/Debug Configurations

No run configurations added.

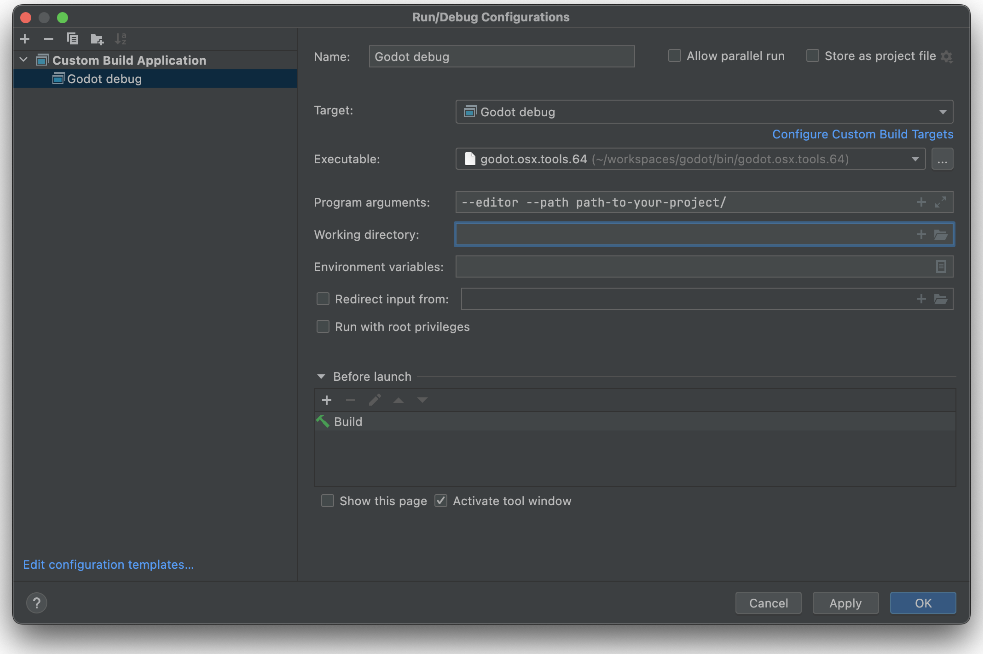
Add new... ^N

Add new run configuration...

or select run configuration to edit

Add New Configuration
Attach to Node.js/Chrome
Boost.Test
Catch
CMake Application
Compound
Core Dump Debug
CTest Application
Custom Build Application
Docker
Docker Image
Docker-compose
Dockerfile
Doctest
Embedded GDB Server
Google Test
Grunt.js
Gulp.js
JavaScript Debug
Jest
Makefile Application
Edit configuration te
Cancel
Apply
OK

- Give the run/debug configuration a name, e.g. `Godot debug`, select the `Godot debug` custom build target as the **Target**. Select the Godot executable in the `bin/` folder as the **Executable**, and set the **Program arguments** to `--editor --path path-to-your-project/`, where `path-to-your-project/` should be a path pointing to an existing Godot project. If you omit the `--path` argument, you will only be able to debug the Godot Project Manager window. Click **OK** to create the run/debug configuration.


Run/Debug Configurations
Custom Build Application
Godot debug
Edit configuration templates...
Cancel
Apply
OK

Name: Godot debug

☐ Allow parallel run

☐ Store as project file

Target: Godot debug

Configure Custom Build Targets

Executable: godot.osx.tools.64 (~/workspaces/godot/bin/godot.osx.tools.64)

Program arguments: `--editor --path path-to-your-project/`

Working directory:

Environment variables:

☐ Redirect input from:

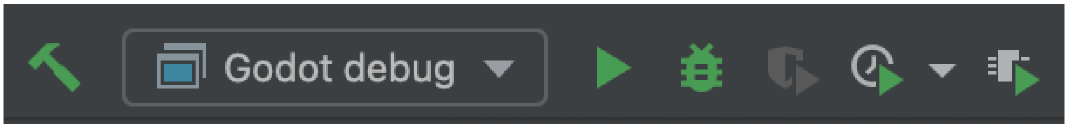
☐ Run with root privileges

Before launch

Build

☐ Show this page ☑ Activate tool window

You can now build, run, debug, profile, and Valgrind check the Godot editor via the run configuration.


Godot debug

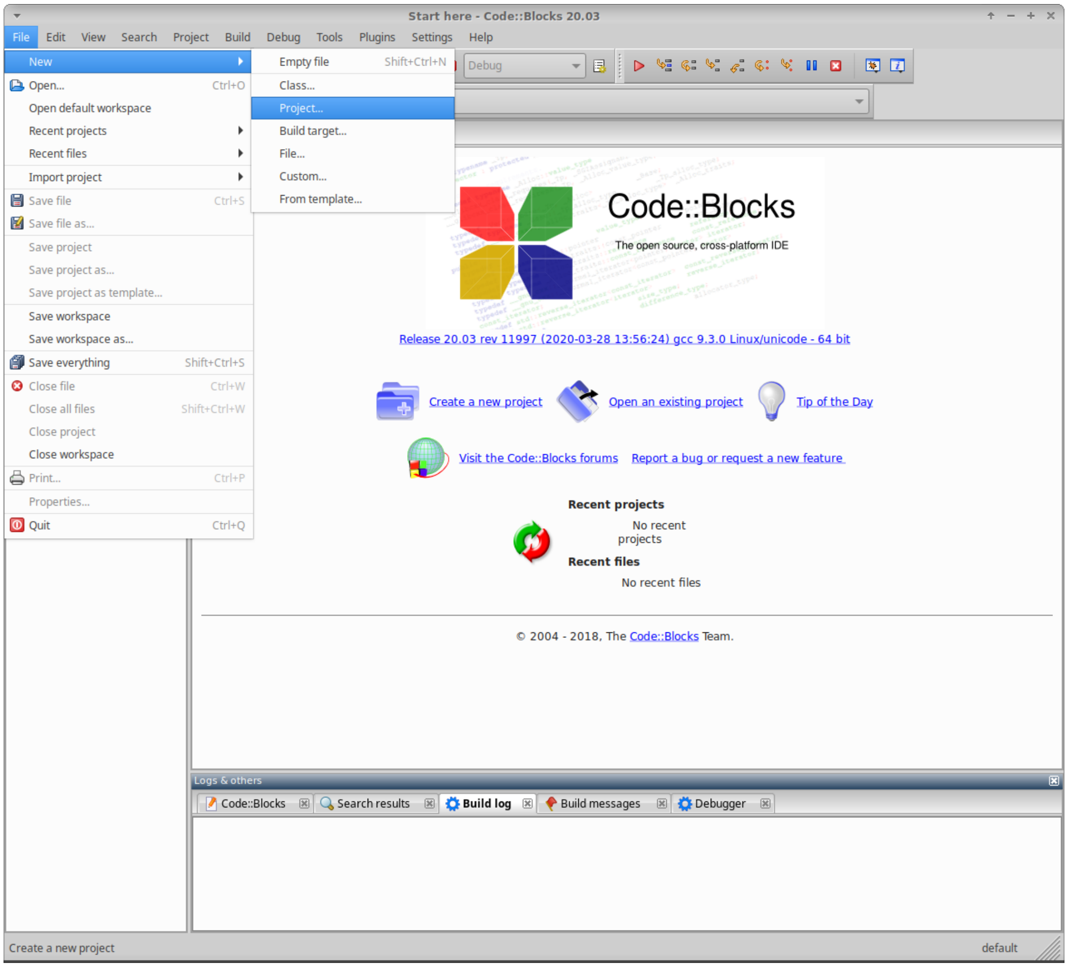
When playing a scene, the Godot editor will spawn a separate process. You can debug this process in CLion by going to **Run > Attach to process...**, typing `godot`, and selecting the Godot process with the highest **pid** (process ID), which will usually be the running project.

Code::Blocks

Code::Blocks [https://codeblocks.org/] is a free, open source, cross platform IDE.

Creating a new project

From Code::Blocks' main screen, click **Create a new project** or select **File > New > Project...**.



Start here - Code::Blocks 20.03
File
Edit
View
Search
Project
Build
Debug
Tools
Plugins
Settings
Help
New
Open... Ctrl+O
Open default workspace
Recent projects
Recent files
Import project
Save file Ctrl+S
Save file as...
Save project
Save project as...
Save project as template...
Save workspace
Save workspace as...
Save everything Shift+Ctrl+S
Close file Ctrl+W
Close all files Shift+Ctrl+W
Close project
Close workspace
Print... Ctrl+P
Properties...
Quit Ctrl+Q
Empty file Shift+Ctrl+N
Class...
Project...
Build target...
File...
Custom...
From template...
Debug
Code::Blocks
The open source, cross-platform IDE
Create a new project
Open an existing project
Tip of the Day
Visit the Code::Blocks forums
Report a bug or request a new feature
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
Create a new project
default

Release 20.03 rev 11997 (2020-03-28 13:56:24) gcc 9.3.0 Linux/unicode - 64 bit

Recent projects

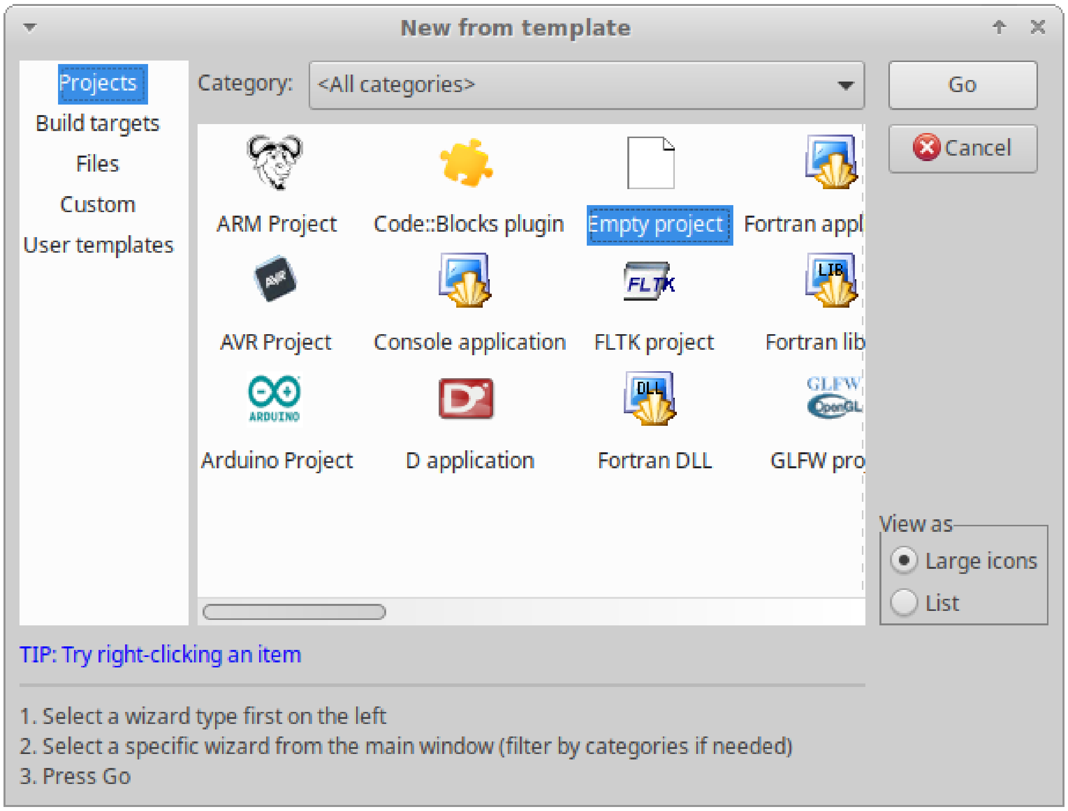
No recent projects

Recent files

No recent files

© 2004 - 2018, The Code::Blocks Team.

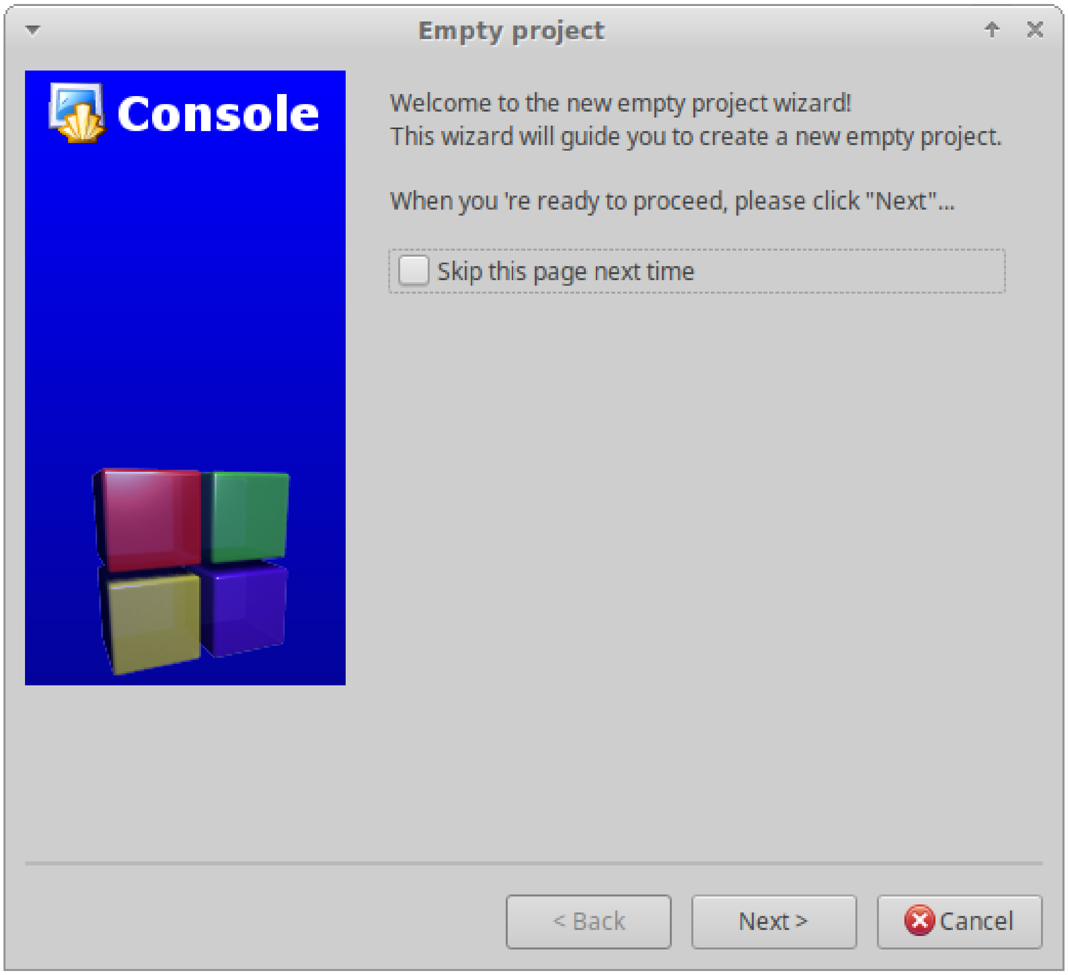
In the **New from template** window, from **Projects**, select **Empty project**, and click **Go**.


New from template
Projects
Build targets
Files
Custom
User templates
Category:
<All categories>
Go
Cancel
ARM Project
Code::Blocks plugin
Empty project
Fortran appl
AVR Project
Console application
FLTK project
Fortran lib
Arduino Project
D application
Fortran DLL
GLFW pro
View as
● Large icons
○ List

TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Click Next, to pass the welcome to the new empty project wizard.



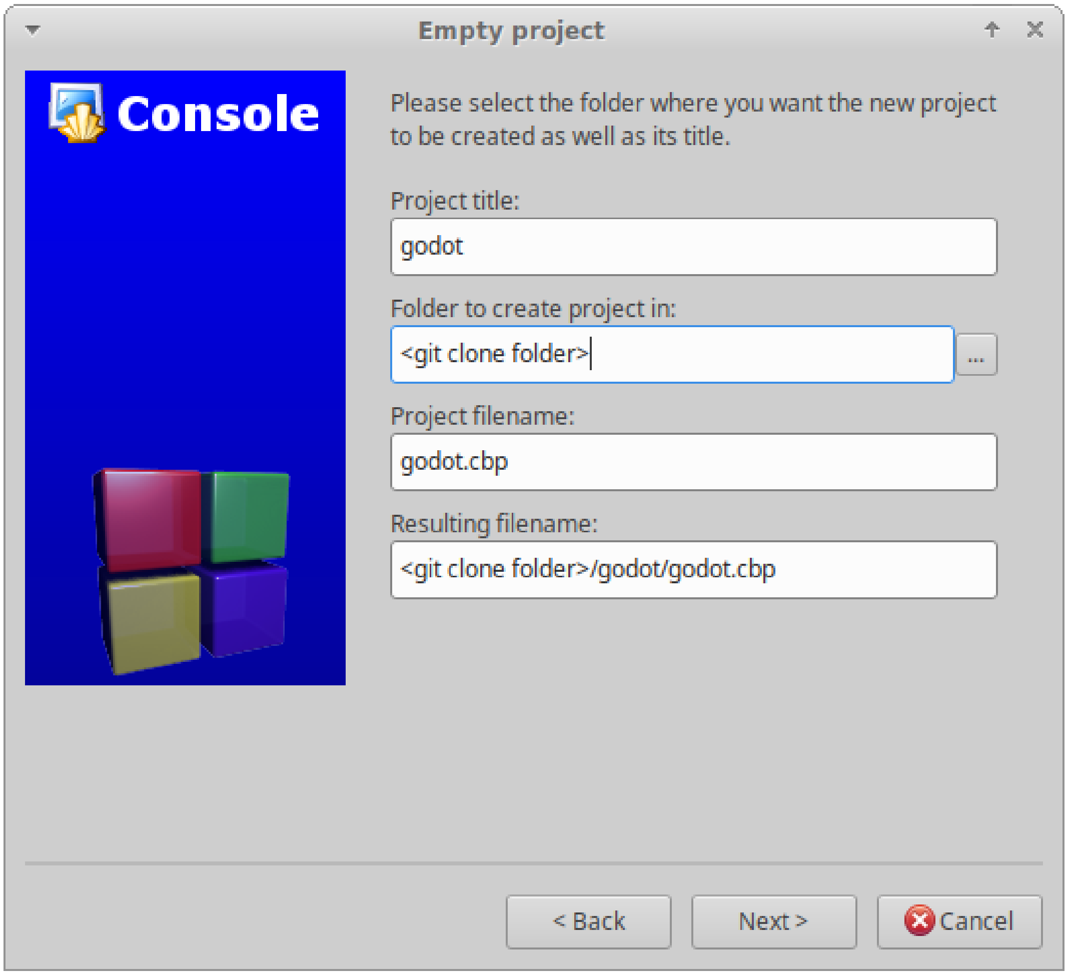
Welcome to the new empty project wizard!
This wizard will guide you to create a new empty project.

When you 're ready to proceed, please click "Next"...

☐ Skip this page next time

The project file should be created in the root of the cloned project folder. To achieve this, first, ensure that the **Project title** is the same as the folder name that Godot was cloned into. Unless you cloned the project into a folder with a different name, this will be `godot`.

Second, ensure that the **Folder to create project in** is the folder you ran the Git clone command from, not the `godot` project folder. Confirm that the **Resulting filename** field will create the project file in the root of the cloned project folder.



Please select the folder where you want the new project to be created as well as its title.

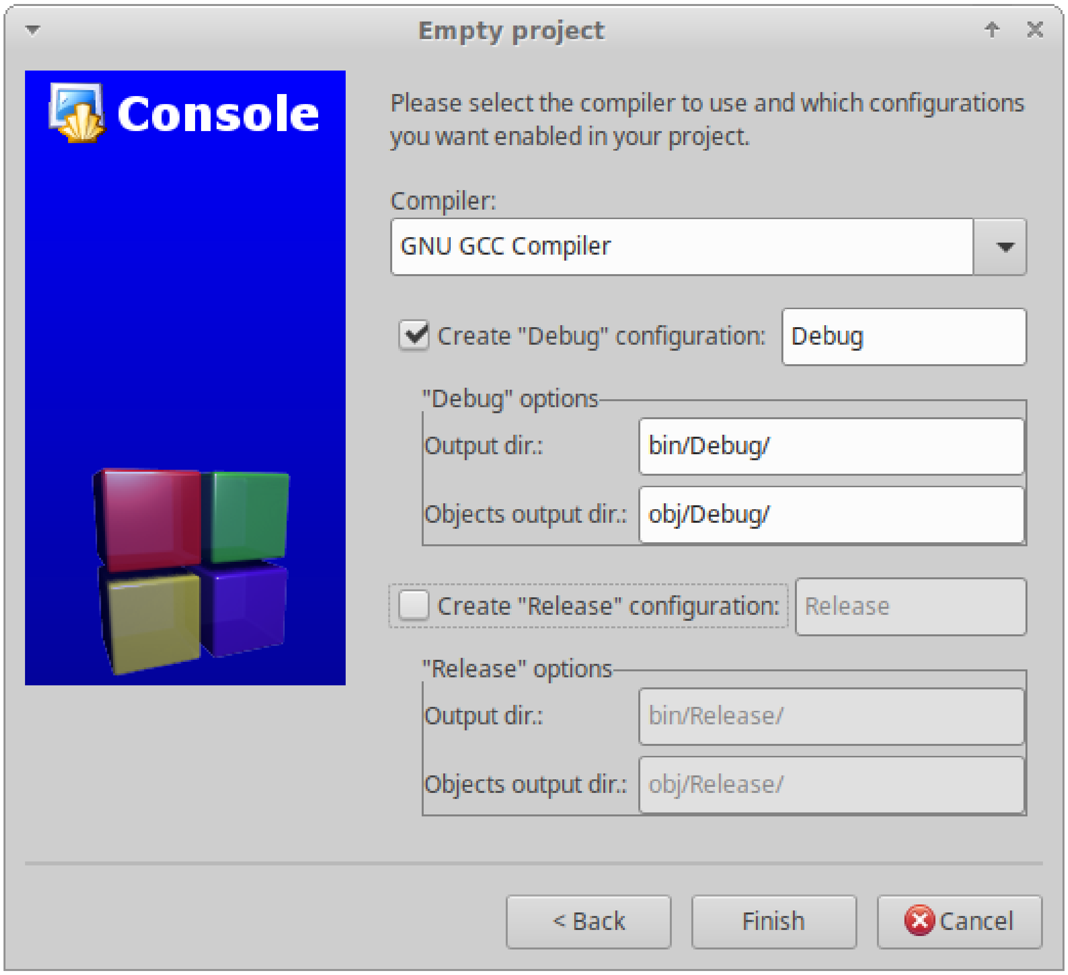
Project title: godot

Folder to create project in: <git clone folder>

Project filename: godot.cbp

Resulting filename: <git clone folder>/godot/godot.cbp

The compiler and configuration settings are managed through **SCons** and will be configured later. However, it's worth deselecting the **Create "Release" configuration** option; so only a single build target is created before clicking **Finish**.



Please select the compiler to use and which configurations you want enabled in your project.

Compiler: GNU GCC Compiler

☑ Create "Debug" configuration: Debug

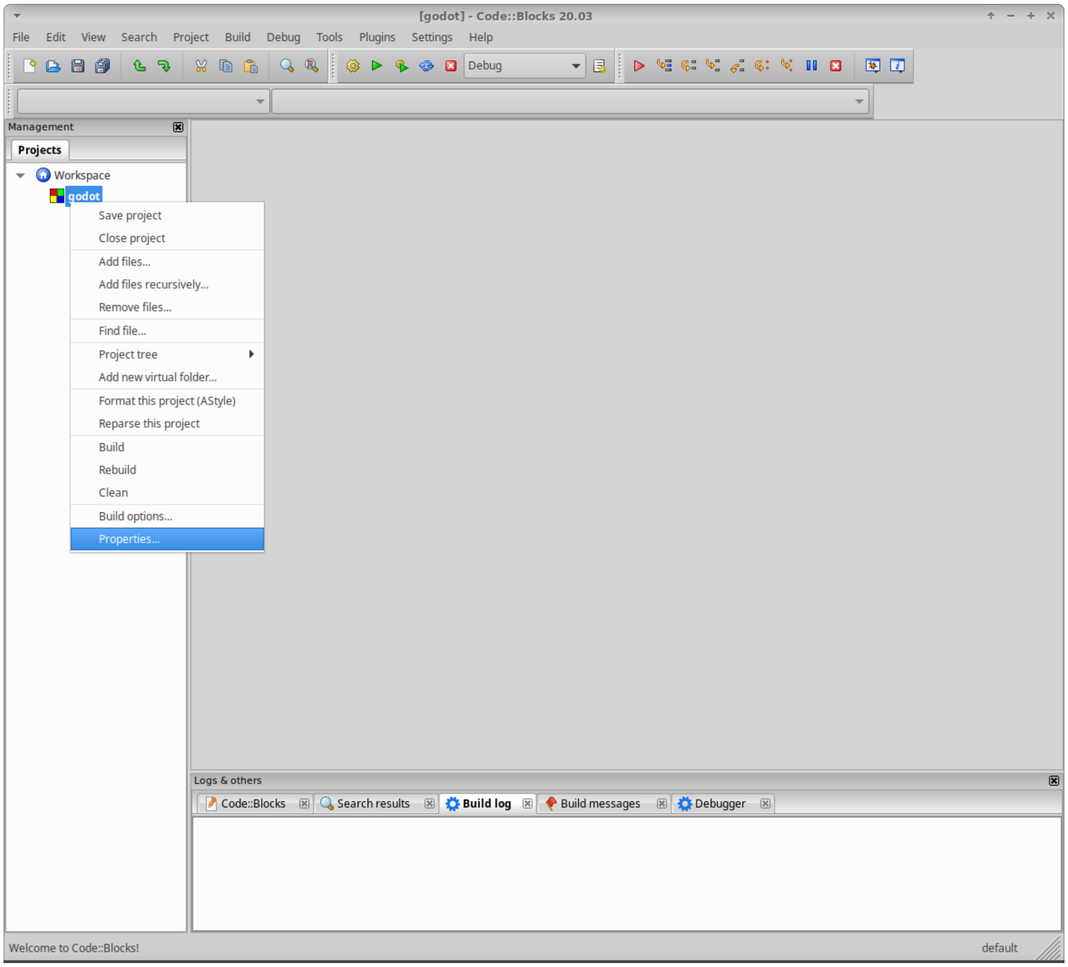
"Debug" options
Output dir.: bin/Debug/
Objects output dir.: obj/Debug/

☐ Create "Release" configuration: Release

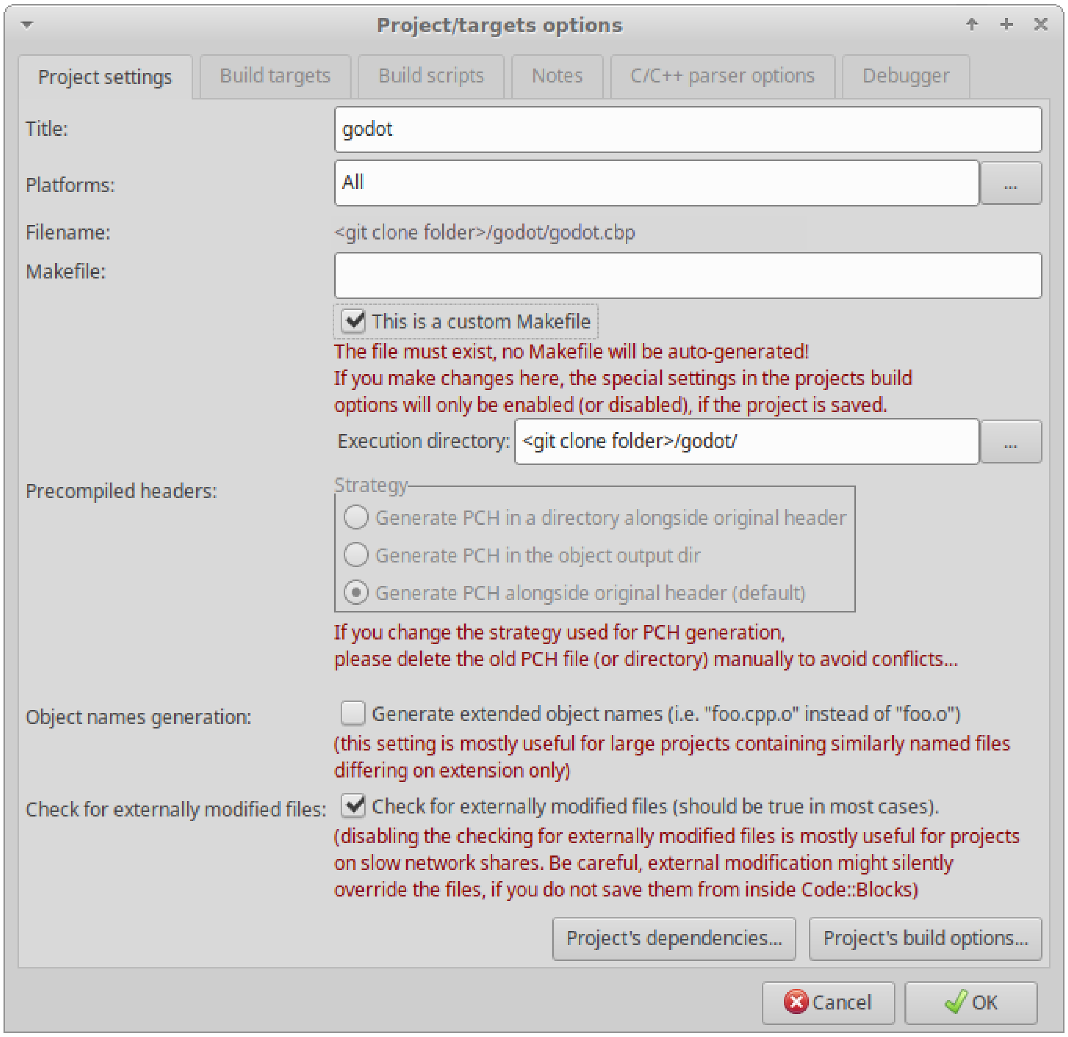
"Release" options
Output dir.: bin/Release/
Objects output dir.: obj/Release/

Configuring the build

The first step is to change the project properties. Right-click on the new project and select **Properties...**.


[godot] - Code::Blocks 20.03
File
Edit
View
Search
Project
Build
Debug
Tools
Plugins
Settings
Help
Debug
Management
Projects
Workspace
godot
Save project
Close project
Add files...
Add files recursively...
Remove files...
Find file...
Project tree
Add new virtual folder...
Format this project (AStyle)
Reparse this project
Build
Rebuild
Clean
Build options...
Properties...
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
Welcome to Code::Blocks!
default

Check the **This is a custom Makefile** property. Click OK to save the changes.


Project/targets options
Project settings
Build targets
Build scripts
Notes
C/C++ parser options
Debugger
...
...
Project's dependencies...
Project's build options...
Cancel
OK

Title: godot

Platforms: All

Filename: <git clone folder>/godot/godot.cbp

Makefile:

☑ This is a custom Makefile

The file must exist, no Makefile will be auto-generated!
If you make changes here, the special settings in the projects build options will only be enabled (or disabled), if the project is saved.

Execution directory: <git clone folder>/godot/

Precompiled headers:

Strategy
○ Generate PCH in a directory alongside original header
○ Generate PCH in the object output dir
● Generate PCH alongside original header (default)

If you change the strategy used for PCH generation, please delete the old PCH file (or directory) manually to avoid conflicts...

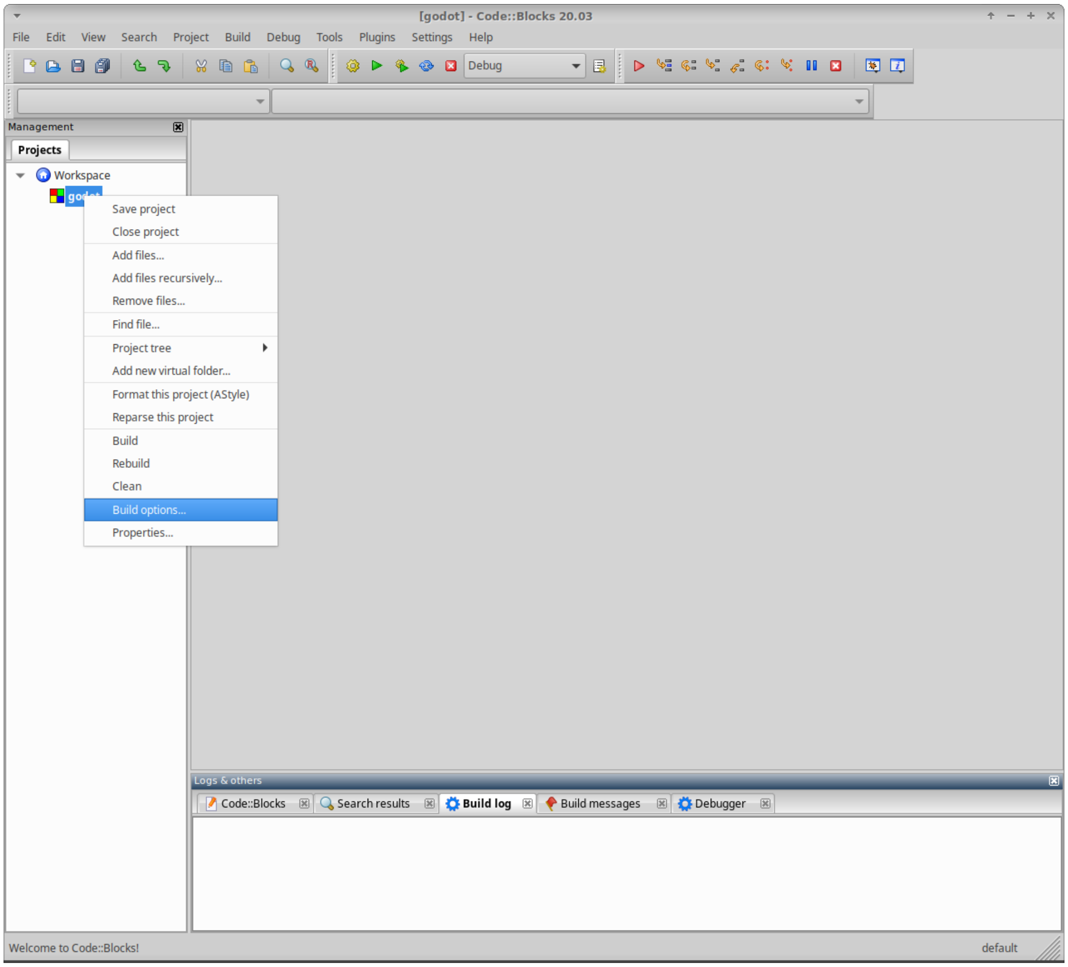
Object names generation: ☐ Generate extended object names (i.e. "foo.cpp.o" instead of "foo.o")

(this setting is mostly useful for large projects containing similarly named files differing on extension only)

Check for externally modified files: ☑ Check for externally modified files (should be true in most cases).

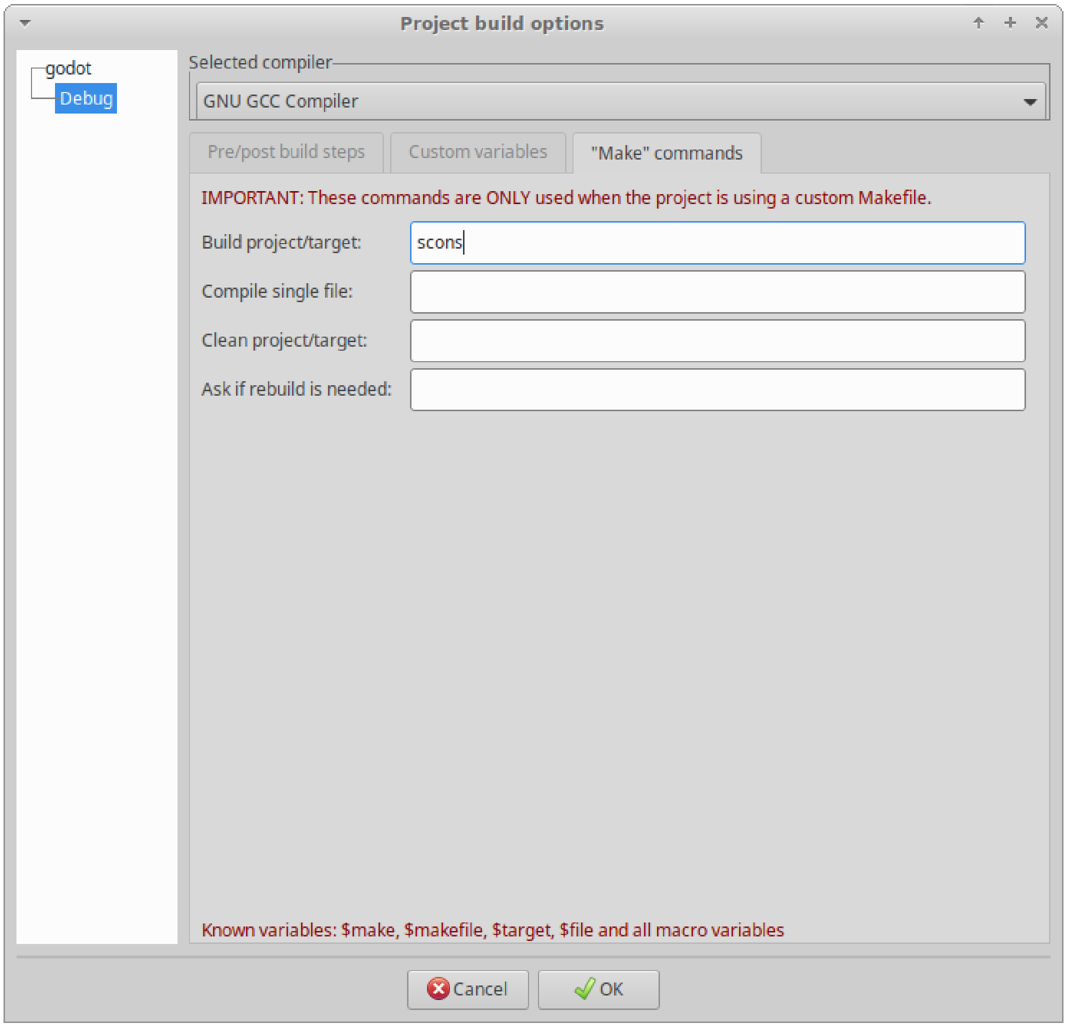
(disabling the checking for externally modified files is mostly useful for projects on slow network shares. Be careful, external modification might silently override the files, if you do not save them from inside Code::Blocks)

The next step is to change the build options. Right-click on the new project and select **Build Options...**.


[godot] - Code::Blocks 20.03
File Edit View Search Project Build Debug Tools Plugins Settings Help
Debug
Management
Projects
Workspace
godot
Save project
Close project
Add files...
Add files recursively...
Remove files...
Find file...
Project tree
Add new virtual folder...
Format this project (AStyle)
Reparse this project
Build
Rebuild
Clean
Build options...
Properties...
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
Welcome to Code::Blocks!
default

Select the **"Make" commands** tab and remove all the existing commands for all the build targets. For each build target enter the **SCons** command for creating the desired build in the **Build project/target** field. The minimum is `scons`. For details on the **SCons** build options, see Introduction to the buildsystem. It's also useful to add the `scons --clean` command in the **Clean project/target** field to the project's default commands.

If you're using Windows, all the commands need to be preceded with `cmd /c` to initialize the command interpreter.



Project build options
godot
Debug
Pre/post build steps
Custom variables
"Make" commands
Cancel
OK

Selected compiler: GNU GCC Compiler

IMPORTANT: These commands are ONLY used when the project is using a custom Makefile.

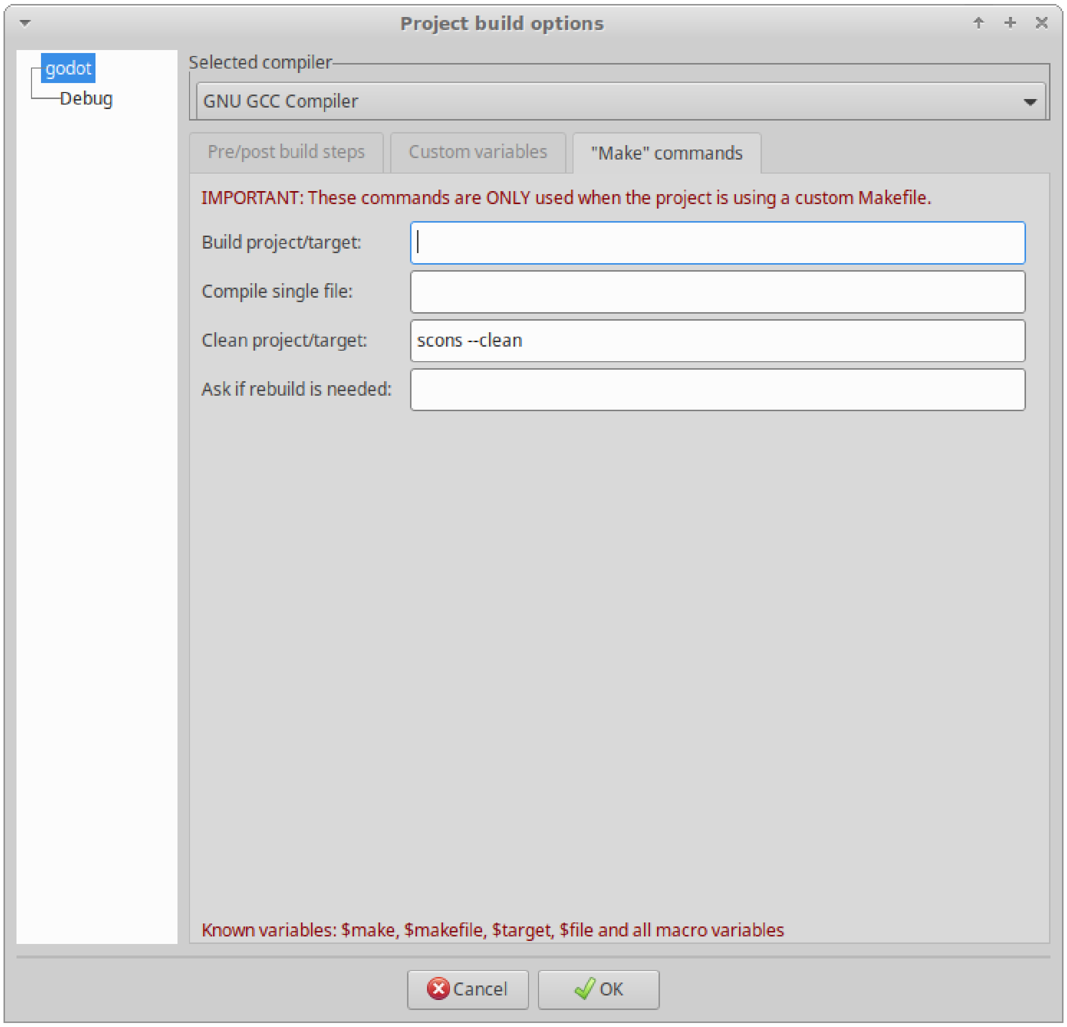
Build project/target: scons

Compile single file:

Clean project/target:

Ask if rebuild is needed:

Known variables: $make, $makefile, $target, $file and all macro variables


Project build options
godot
Debug
Pre/post build steps
Custom variables
"Make" commands
Cancel
OK

Selected compiler: GNU GCC Compiler

IMPORTANT: These commands are ONLY used when the project is using a custom Makefile.

Build project/target:

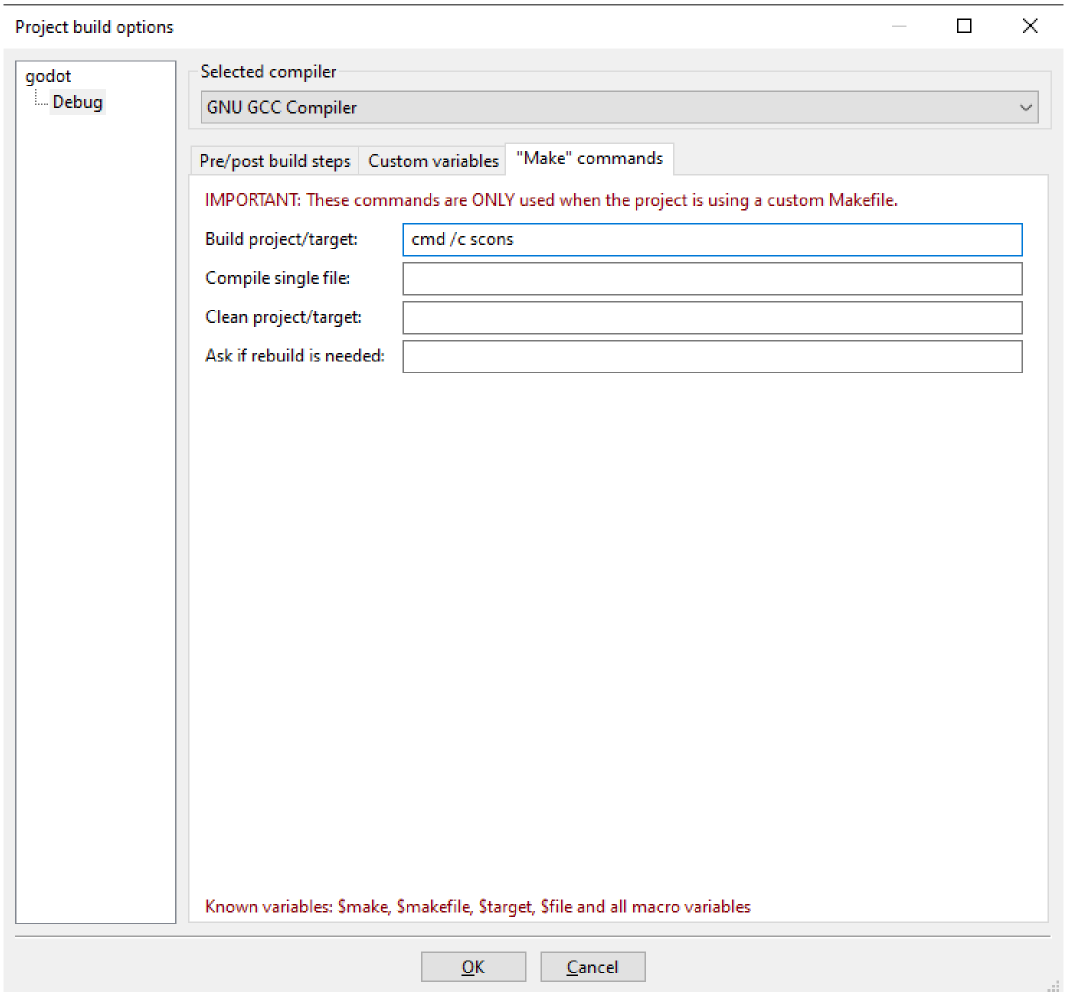
Compile single file:

Clean project/target: scons --clean

Ask if rebuild is needed:

Known variables: $make, $makefile, $target, $file and all macro variables

Windows example:


Project build options
godot
Debug
Pre/post build steps
Custom variables
"Make" commands
OK
Cancel

Selected compiler: GNU GCC Compiler

IMPORTANT: These commands are ONLY used when the project is using a custom Makefile.

Build project/target: cmd /c scons

Compile single file:

Clean project/target:

Ask if rebuild is needed:

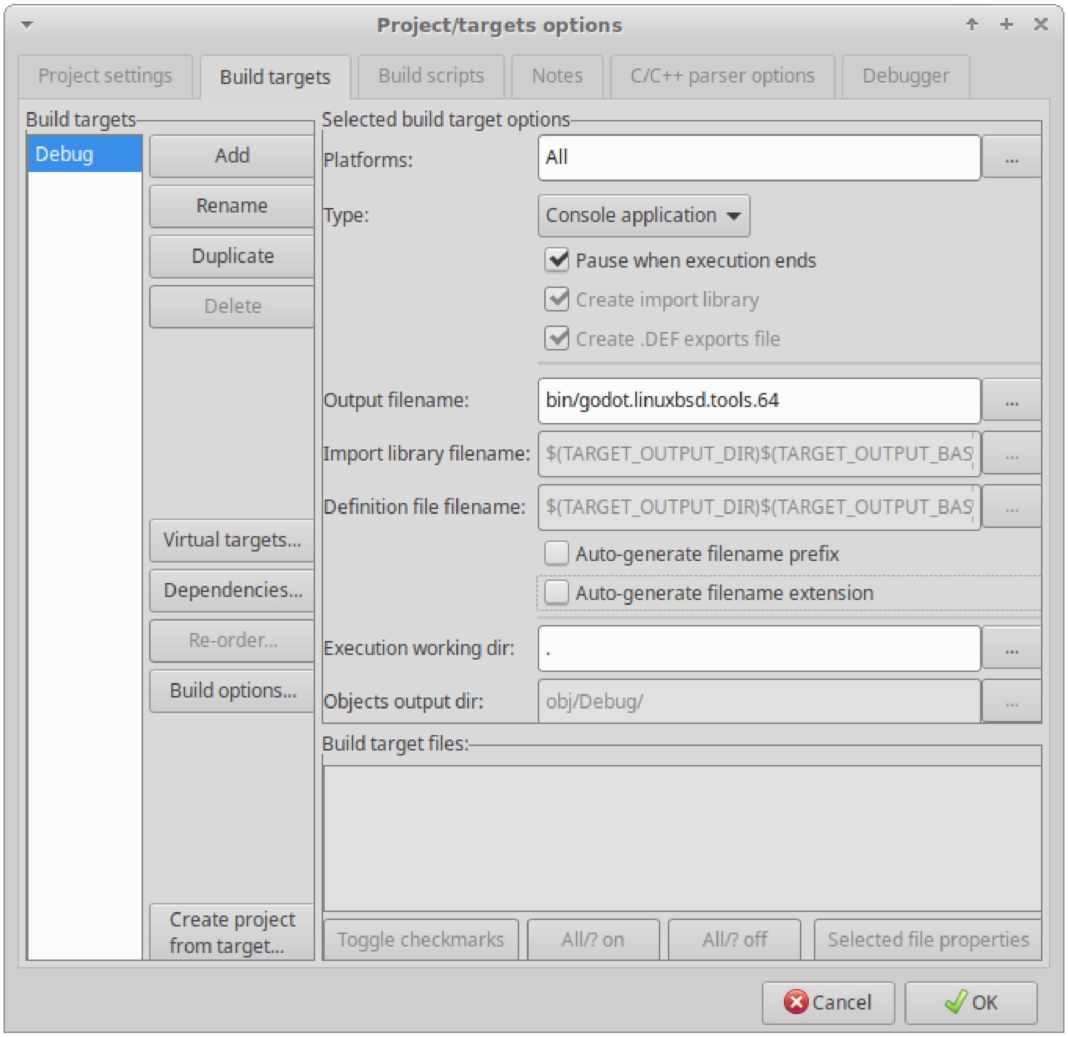
Known variables: $make, $makefile, $target, $file and all macro variables

Code::Blocks should now be configured to build Godot; so either select **Build > Build**, click the gear button, or press `Ctrl + F9`.

Configuring the run

Once **SCons** has successfully built the desired target, reopen the project **Properties...** and select the **Build targets** tab. In the **Output filename** field, browse to the `bin` folder and select the compiled file.

Deselect the **Auto-generate filename prefix** and **Auto-generate filename extension** options.



Project/targets options
Project settings
Build targets
Build scripts
Notes
C/C++ parser options
Debugger
Add
Rename
Duplicate
Delete
Virtual targets...
Dependencies...
Re-order...
Build options...
Create project from target...
Toggle checkmarks
All/? on
All/? off
Selected file properties
Cancel
OK

Build targets: Debug

Selected build target options

Platforms: All

Type: Console application

☑ Pause when execution ends

☑ Create import library

☑ Create .DEF exports file

Output filename: bin/godot.linuxbsd.tools.64

Import library filename: $(TARGET_OUTPUT_DIR)$(TARGET_OUTPUT_BAS

Definition file filename: $(TARGET_OUTPUT_DIR)$(TARGET_OUTPUT_BAS

☐ Auto-generate filename prefix

☐ Auto-generate filename extension

Execution working dir: .

Objects output dir: obj/Debug/

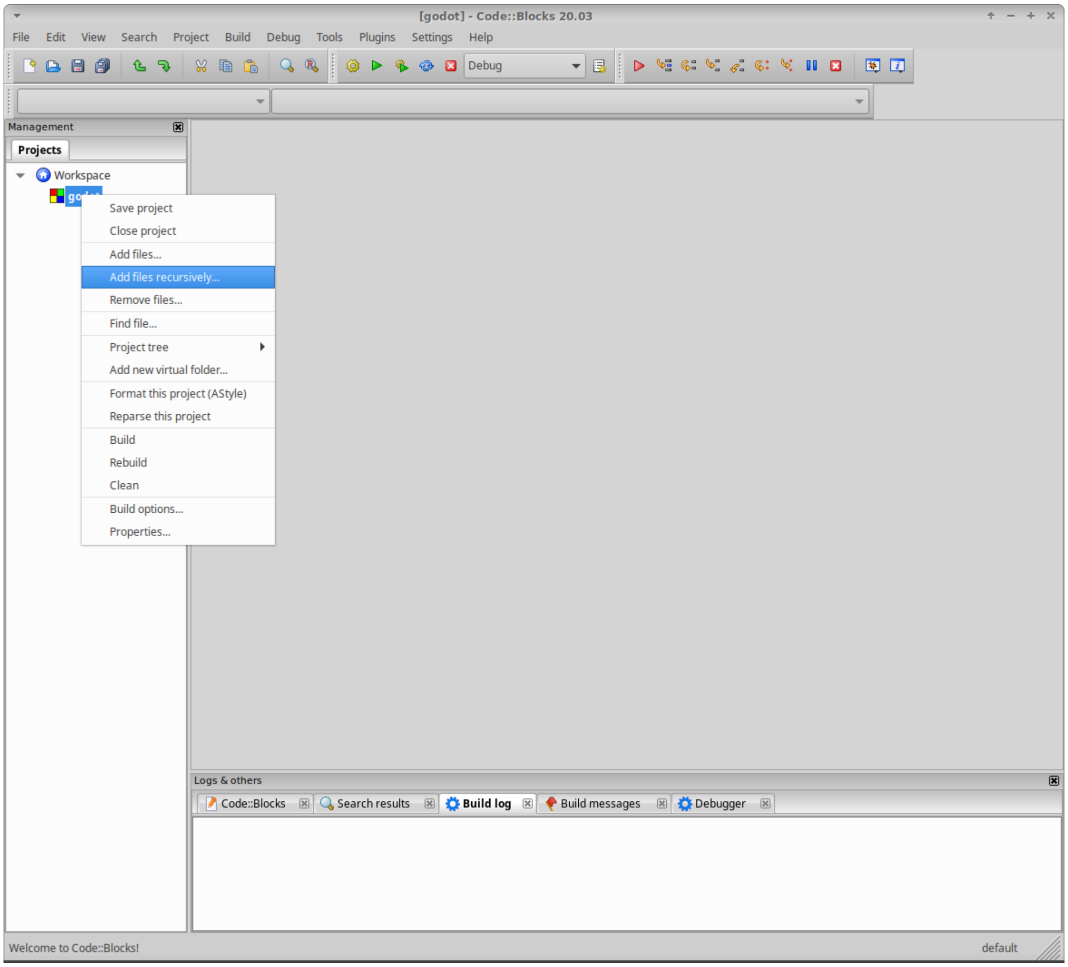
Build target files:

Code::Blocks should now be configured to run your compiled Godot executable; so either select **Build > Run**, click the green arrow button, or press `Ctrl + F10`.

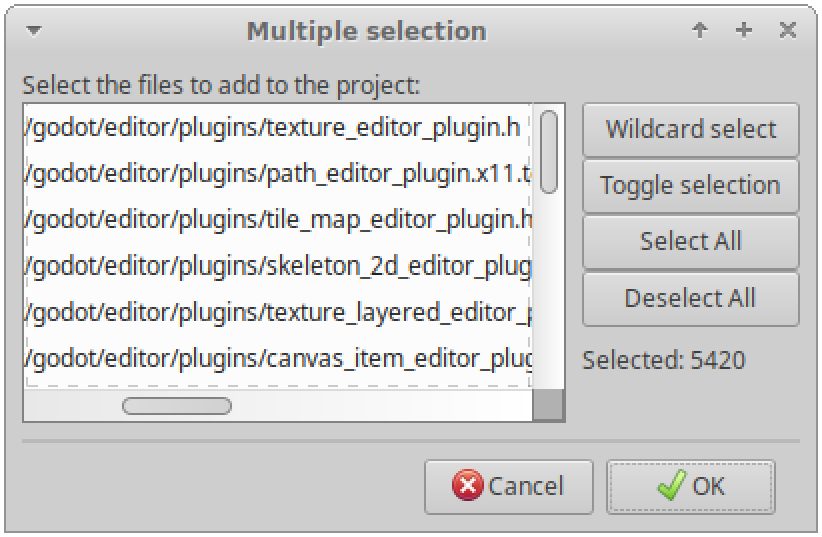
There are two additional points worth noting. First, if required, the **Execution working dir** field can be used to test specific projects, by setting it to the folder containing the `project.godot` file. Second, the **Build targets** tab can be used to add and remove build targets for working with and creating different builds.

Adding files to the project

To add all the Godot code files to the project, right-click on the new project and select **Add files recursively...**.


[godot] - Code::Blocks 20.03
File
Edit
View
Search
Project
Build
Debug
Tools
Plugins
Settings
Help
Debug
Management
Projects
Workspace
godot
Save project
Close project
Add files...
Add files recursively...
Remove files...
Find file...
Project tree
Add new virtual folder...
Format this project (AStyle)
Reparse this project
Build
Rebuild
Clean
Build options...
Properties...
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
Welcome to Code::Blocks!
default

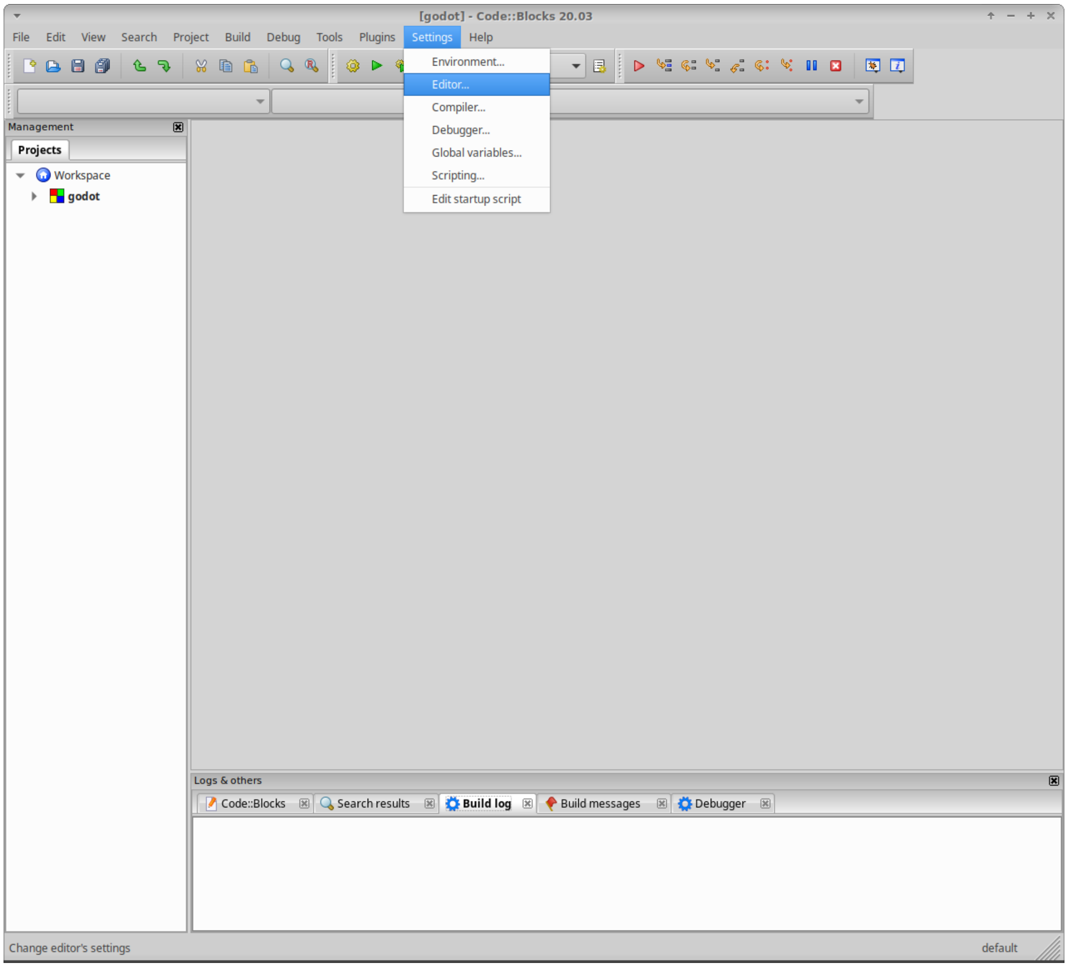
It should automatically select the project folder; so simply click **Open**. By default, all code files are included, so simply click **OK**.



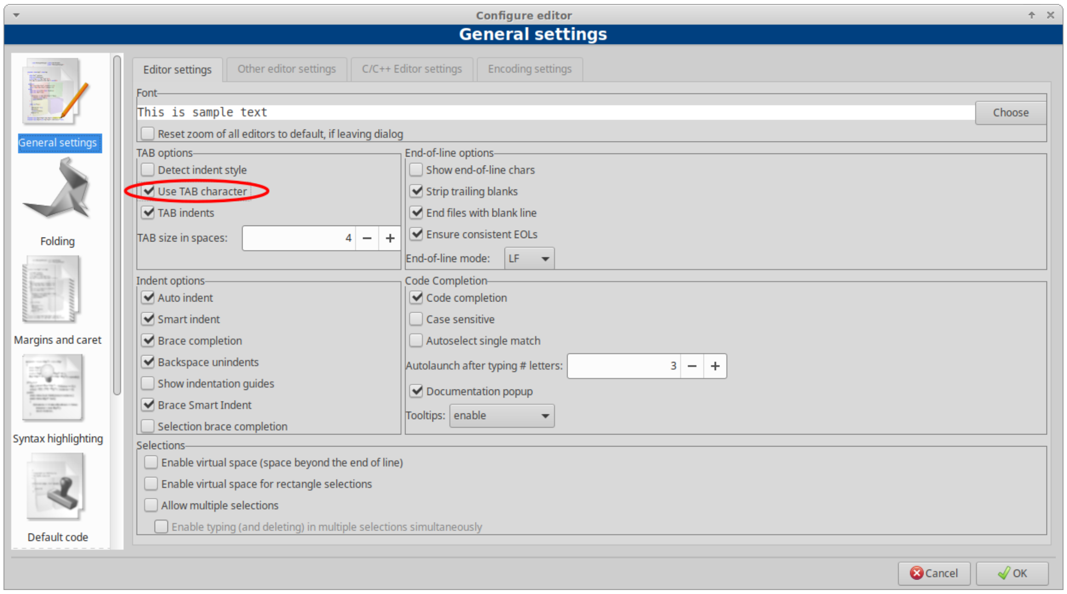
Multiple selection
Select the files to add to the project:
/godot/editor/plugins/texture_editor_plugin.h
/godot/editor/plugins/path_editor_plugin.x11.t
/godot/editor/plugins/tile_map_editor_plugin.h
/godot/editor/plugins/skeleton_2d_editor_plug
/godot/editor/plugins/texture_layered_editor_p
/godot/editor/plugins/canvas_item_editor_plug
Wildcard select
Toggle selection
Select All
Deselect All
Selected: 5420
Cancel
OK

Code style configuration

Before editing any files, remember that all code needs to comply with the Code style guidelines. One important difference with Godot is the use of tabs for indents. Therefore, the key default editor setting that needs to be changed in Code::Blocks is to enable tabs for indents. This setting can be found by selecting **Settings > Editor**.


[godot] - Code::Blocks 20.03
File
Edit
View
Search
Project
Build
Debug
Tools
Plugins
Settings
Help
Environment...
Editor...
Compiler...
Debugger...
Global variables...
Scripting...
Edit startup script
Management
Projects
Workspace
godot
Logs & others
Code::Blocks
Search results
Build log
Build messages
Debugger
Change editor's settings
default

Under **General Settings**, on the **Editor Settings** tab, under **Tab Options** check **Use TAB character**.


Configure editor
General settings
General settings
Folding
Margins and caret
Syntax highlighting
Default code
Editor settings
Other editor settings
C/C++ Editor settings
Encoding settings
Choose
Cancel
OK

Font

This is sample text

☐ Reset zoom of all editors to default, if leaving dialog

TAB options

☐ Detect indent style
☑ Use TAB character
☑ TAB indents
TAB size in spaces: 4

End-of-line options

☐ Show end-of-line chars
☑ Strip trailing blanks
☑ End files with blank line
☑ Ensure consistent EOLs
End-of-line mode: LF

Indent options

☑ Auto indent
☑ Smart indent
☑ Brace completion
☑ Backspace unindents
☐ Show indentation guides
☑ Brace Smart Indent
☐ Selection brace completion

Code Completion

☑ Code completion
☐ Case sensitive
☐ Autoselect single match
Autolaunch after typing # letters: 3
☑ Documentation popup
Tooltips: enable

Selections

☐ Enable virtual space (space beyond the end of line)
☐ Enable virtual space for rectangle selections
☐ Allow multiple selections
☐ Enable typing (and deleting) in multiple selections simultaneously

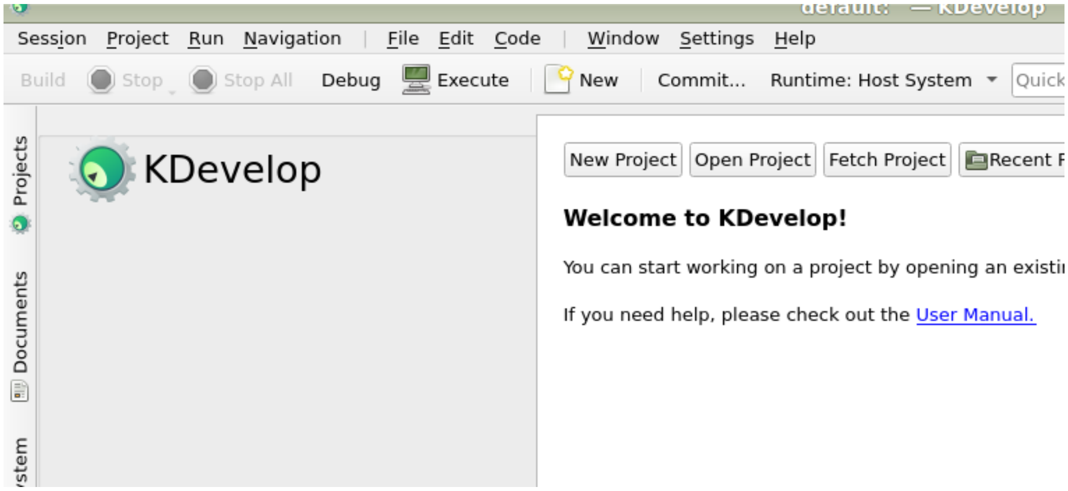
That's it. You're ready to start contributing to Godot using the Code::Blocks IDE. Remember to save the project file and the **Workspace**. If you run into any issues, ask for help in one of Godot's community channels [https://godotengine.org/community].

KDevelop

[KDevelop](https://www.kdevelop.org) [https://www.kdevelop.org] is a free, open source IDE for all desktop platforms.

Importing the project

- From the KDevelop's main screen select **Open Project**.


default: — KDevelop
Session Project Run Navigation File Edit Code Window Settings Help
Build Stop Stop All Debug Execute New Commit... Runtime: Host System Quick
Projects
Documents
stem
KDevelop
New Project
Open Project
Fetch Project
Recent P

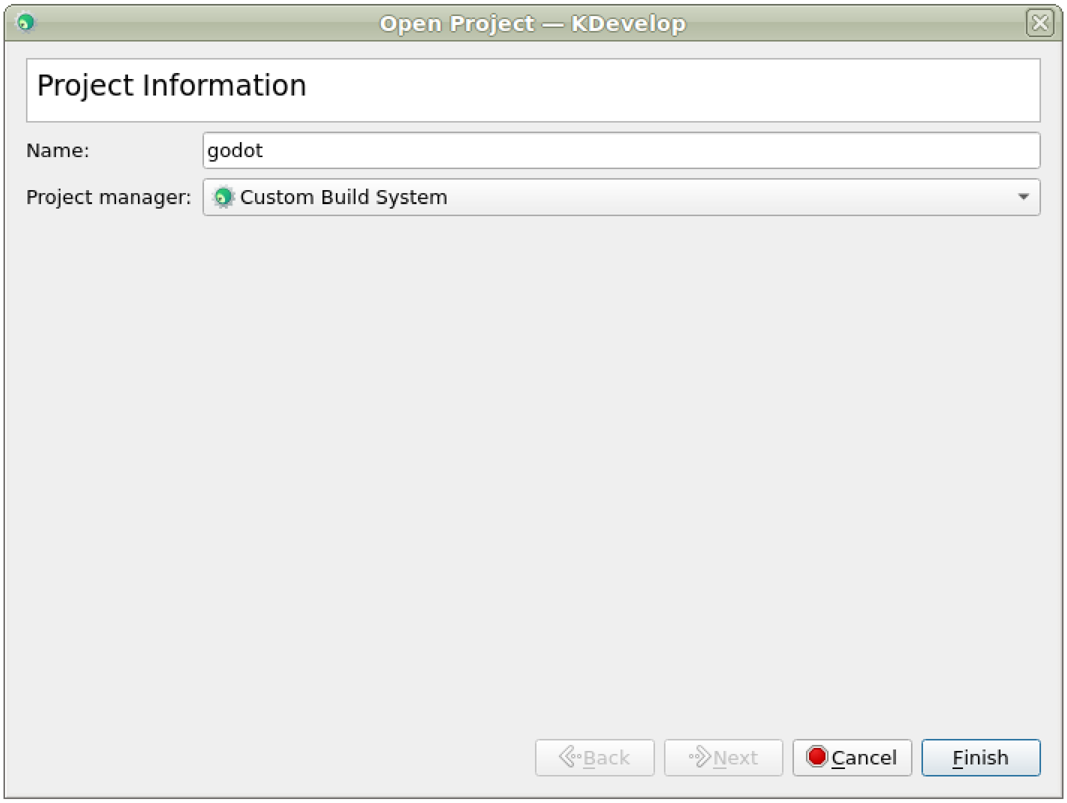
Welcome to KDevelop!

You can start working on a project by opening an existi

If you need help, please check out the User Manual.

KDevelop's main screen.

- Navigate to the Godot root folder and select it.
- On the next screen, choose **Custom Build System** for the **Project Manager**.

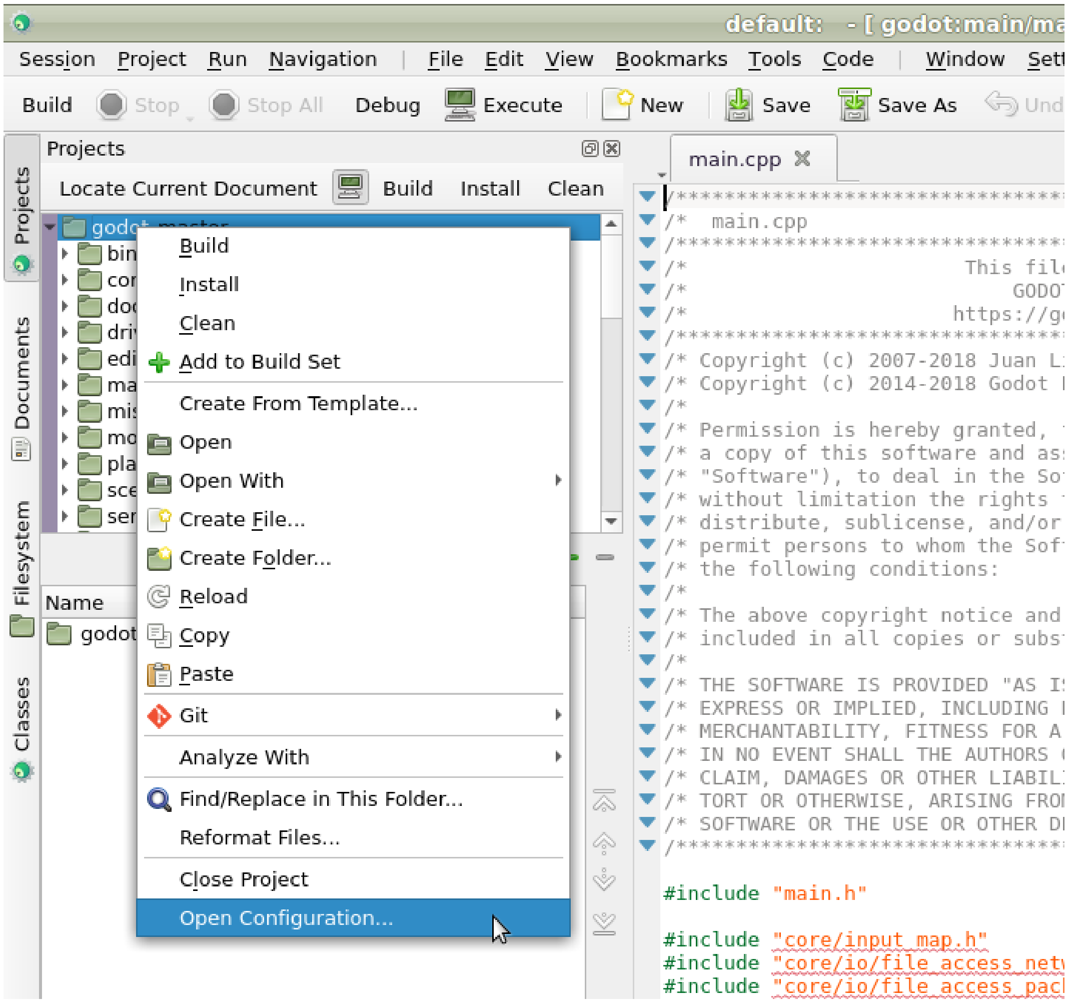

Open Project — KDevelop
Back
Next
Cancel
Finish

Project Information

Name: godot

Project manager: Custom Build System

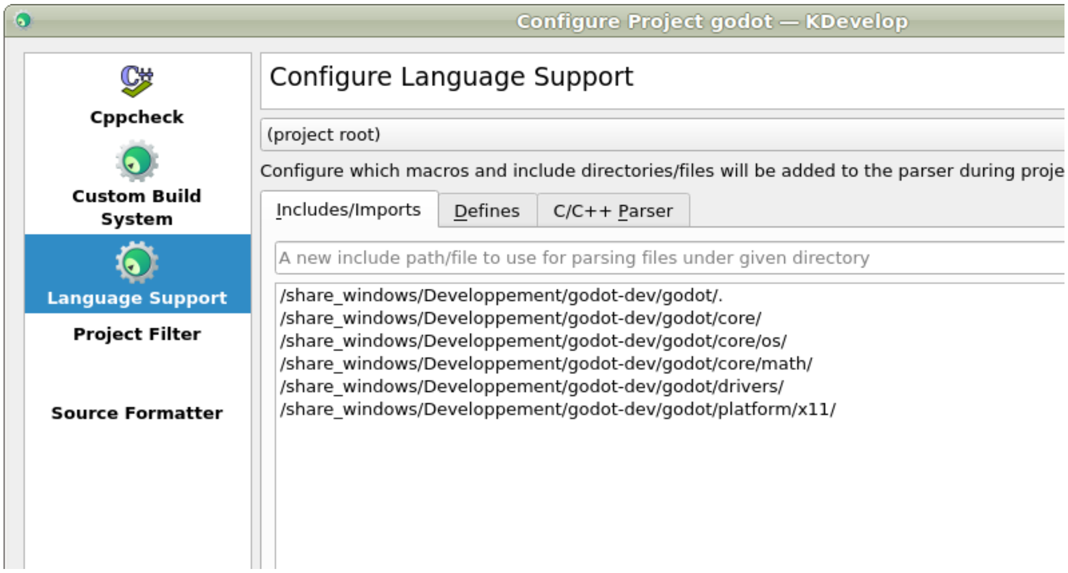
- After the project has been imported, open the project configuration by right-clicking on it in the **Projects** panel and selecting **Open Configuration..** option.


default: - [godot:main/ma
Session
Project
Run
Navigation
File
Edit
View
Bookmarks
Tools
Code
Window
Sett
Build
Stop
Stop All
Debug
Execute
New
Save
Save As
Und
Projects
Locate Current Document
Build
Install
Clean
godot-master
bin
cor
doc
dri
edi
ma
mis
mo
pla
sce
ser
Name
godot
Projects
Documents
Filesystem
Classes
main.cpp
Build
Install
Clean
Add to Build Set
Create From Template...
Open
Open With
Create File...
Create Folder...
Reload
Copy
Paste
Git
Analyze With
Find/Replace in This Folder...
Reformat Files...
Close Project
Open Configuration...

```
/*********************************
/*  main.cpp
/*********************************
/*                          This fil
/*                              GODO
/*                         https://g
/*********************************
/* Copyright (c) 2007-2018 Juan L
/* Copyright (c) 2014-2018 Godot I
/*
/* Permission is hereby granted,
/* a copy of this software and as
/* "Software"), to deal in the So
/* without limitation the rights
/* distribute, sublicense, and/or
/* permit persons to whom the Sof
/* the following conditions:
/*
/* The above copyright notice and
/* included in all copies or subs
/*
/* THE SOFTWARE IS PROVIDED "AS I
/* EXPRESS OR IMPLIED, INCLUDING
/* MERCHANTABILITY, FITNESS FOR A
/* IN NO EVENT SHALL THE AUTHORS
/* CLAIM, DAMAGES OR OTHER LIABIL
/* TORT OR OTHERWISE, ARISING FRO
/* SOFTWARE OR THE USE OR OTHER D
/*********************************

#include "main.h"

#include "core/input_map.h"
#include "core/io/file_access_netw
#include "core/io/file_access_pac
```

- Under **Language Support** open the **Includes/Imports** tab and add the following paths:

```
.  // A dot, to indicate the root of the Godot project
core/
core/os/
core/math/
drivers/
platform/<your_platform>/  // Replace <your_platform> with
a folder
                              corresponding to your current
platform
```


Configure Project godot — KDevelop
Cppcheck
Custom Build System
Language Support
Project Filter
Source Formatter
Includes/Imports
Defines
C/C++ Parser

Configure Language Support

(project root)

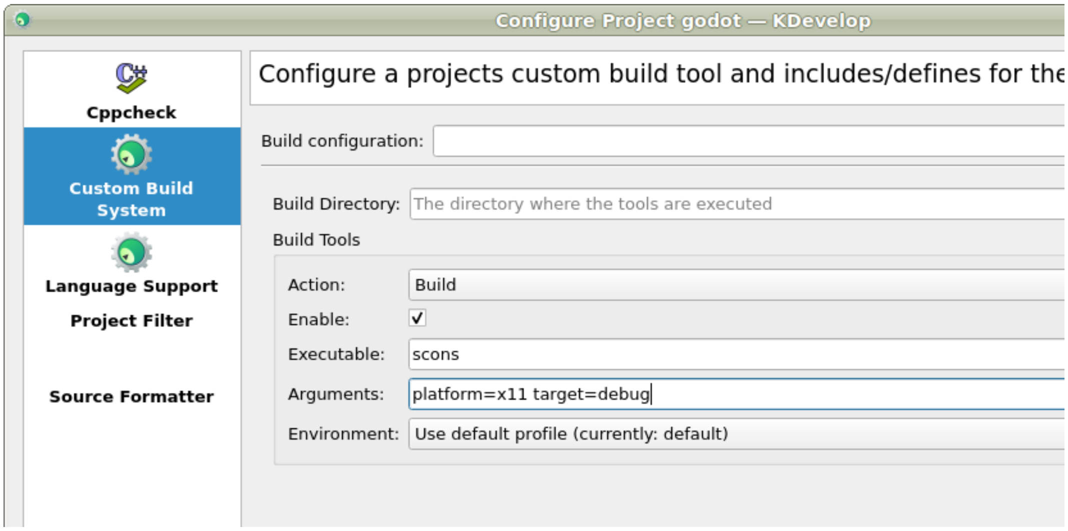
Configure which macros and include directories/files will be added to the parser during proje

A new include path/file to use for parsing files under given directory

```
/share_windows/Developpement/godot-dev/godot/.
/share_windows/Developpement/godot-dev/godot/core/
/share_windows/Developpement/godot-dev/godot/core/os/
/share_windows/Developpement/godot-dev/godot/core/math/
/share_windows/Developpement/godot-dev/godot/drivers/
/share_windows/Developpement/godot-dev/godot/platform/x11/
```

- Apply the changes.
- Under **Custom Build System** add a new build configuration with the following settings:

Build Directory	*blank*
Enable	**True**
Executable	**scons**
Arguments	See Introduction to the buildsystem for a full list of arguments.


Configure Project godot — KDevelop
Cppcheck
Custom Build System
Language Support
Project Filter
Source Formatter

Configure a projects custom build tool and includes/defines for the

Build configuration:

Build Directory: The directory where the tools are executed

Build Tools

Action: Build

Enable: ☑

Executable: scons

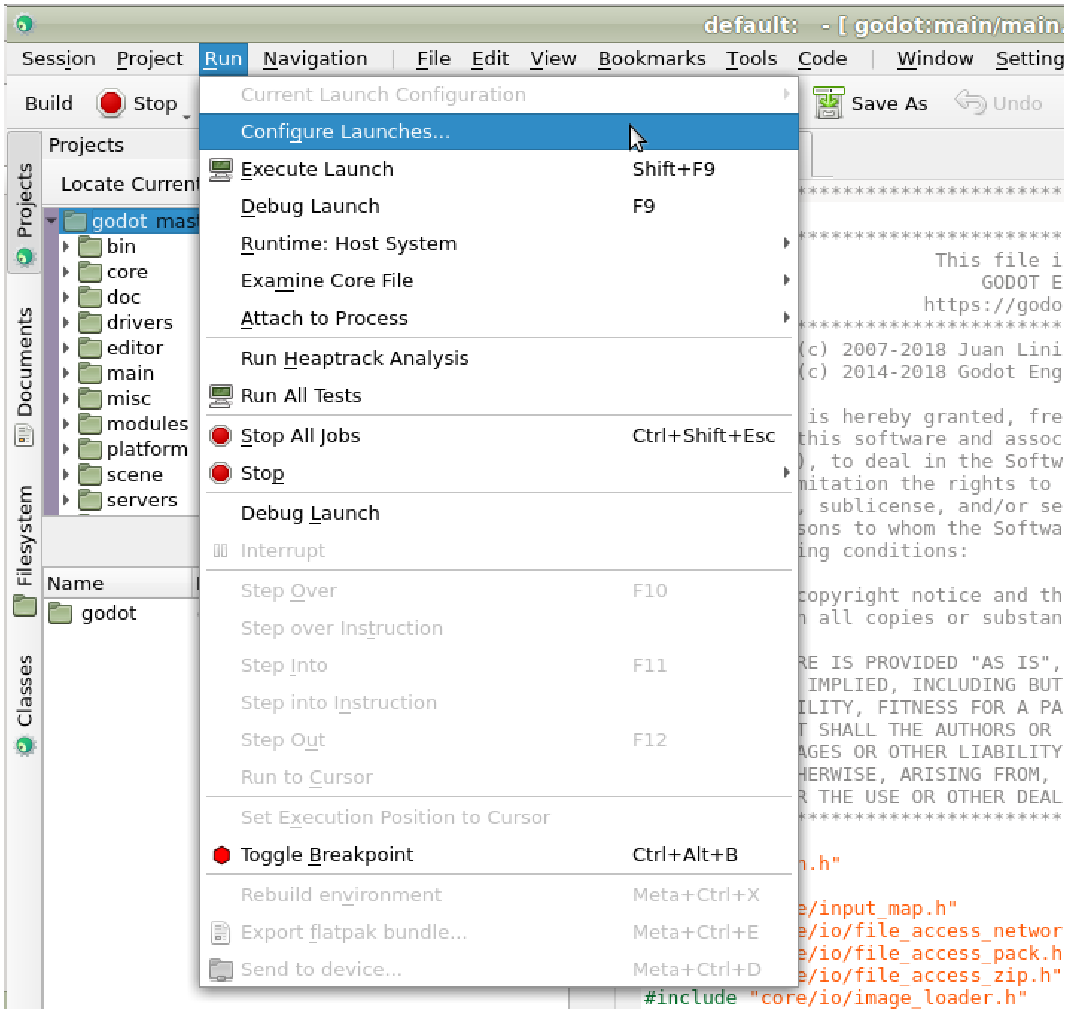
Arguments: platform=x11 target=debug

Environment: Use default profile (currently: default)

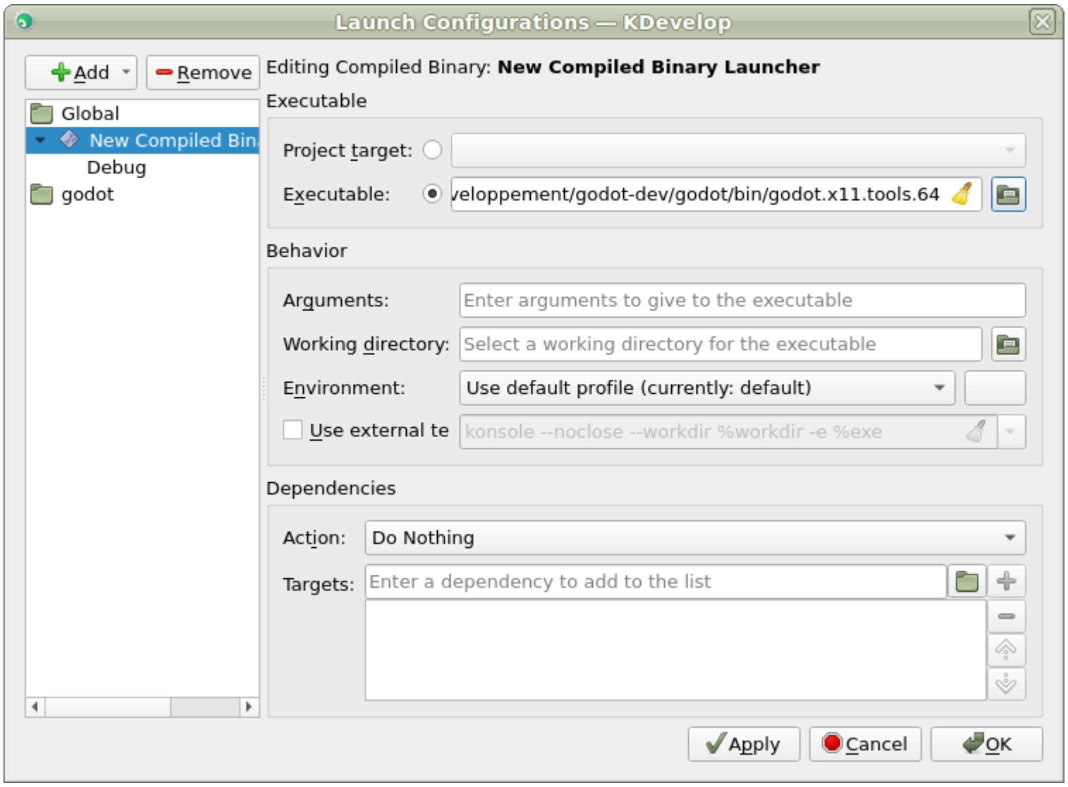
- Apply the changes and close the configuration window.

Debugging the project

- Select **Run > Configure Launches...** from the top menu.


default: - [godot:main/main.
Session
Project
Run
Navigation
File
Edit
View
Bookmarks
Tools
Code
Window
Setting
Build
Stop
Save As
Undo
Projects
Locate Current
godot mas
bin
core
doc
drivers
editor
main
misc
modules
platform
scene
servers
Name
godot
Projects
Documents
Filesystem
Classes
Current Launch Configuration
Configure Launches...
Execute Launch Shift+F9
Debug Launch F9
Runtime: Host System
Examine Core File
Attach to Process
Run Heaptrack Analysis
Run All Tests
Stop All Jobs Ctrl+Shift+Esc
Stop
Debug Launch
Interrupt
Step Over F10
Step over Instruction
Step Into F11
Step into Instruction
Step Out F12
Run to Cursor
Set Execution Position to Cursor
Toggle Breakpoint Ctrl+Alt+B
Rebuild environment Meta+Ctrl+X
Export flatpak bundle... Meta+Ctrl+E
Send to device... Meta+Ctrl+D

- Click **Add** to create a new launch configuration.
- Select **Executable** option and specify the path to your executable located in the `<Godot root directory>/bin` folder. The name depends on your build configuration, e.g. `godot.linuxbsd.editor.dev.x86_64` for 64-bit LinuxBSD platform with `platform=editor` and `dev_build=yes`.



Launch Configurations — KDevelop
Add
Remove
Global
New Compiled Bin
Debug
godot
Executable
Behavior
Dependencies
Apply
Cancel
OK

Editing Compiled Binary: **New Compiled Binary Launcher**

Project target: ○

Executable: ● veloppement/godot-dev/godot/bin/godot.x11.tools.64

Arguments: Enter arguments to give to the executable

Working directory: Select a working directory for the executable

Environment: Use default profile (currently: default)

☐ Use external te konsole --noclose --workdir %workdir -e %exe

Action: Do Nothing

Targets: Enter a dependency to add to the list

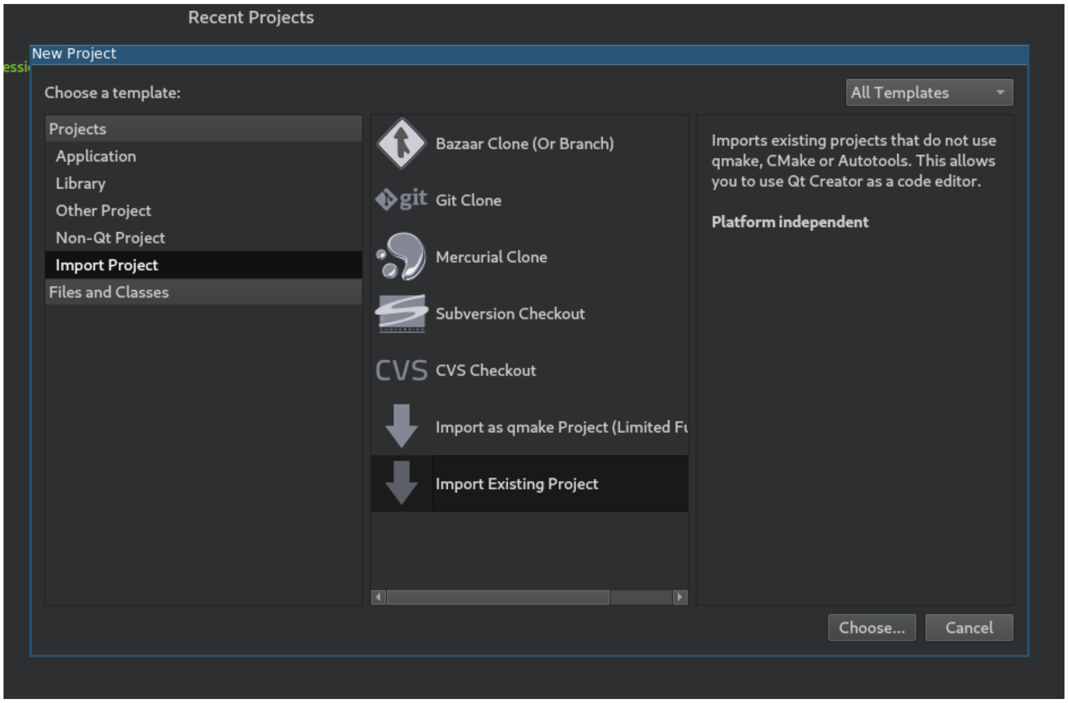
If you run into any issues, ask for help in one of Godot's community channels [https://godotengine.org/community].

Qt Creator

Qt Creator [https://doc.qt.io/qtcreator/index.html] is a free, open source IDE for all desktop platforms.

Importing the project

- From the Qt Creator's main screen select **New Project > Import Project > Import Existing Project**.

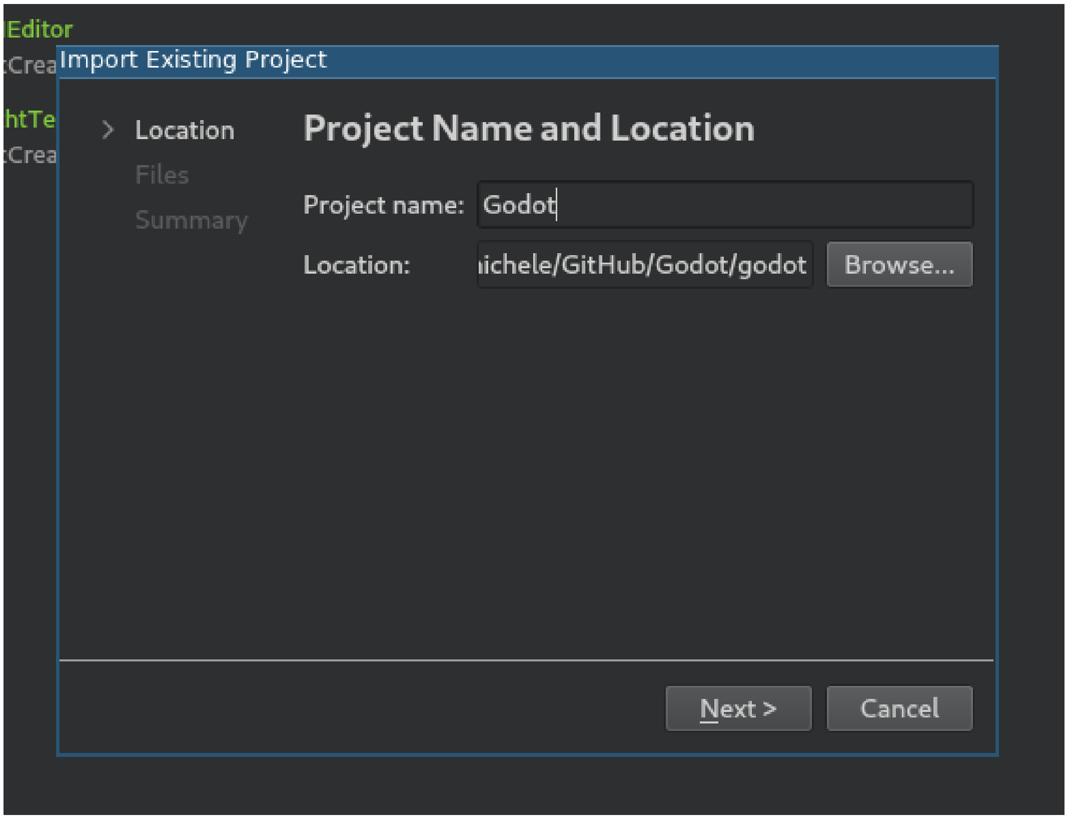


Recent Projects
New Project
Choose a template:
All Templates
Projects
Application
Library
Other Project
Non-Qt Project
Import Project
Files and Classes
Bazaar Clone (Or Branch)
git Git Clone
Mercurial Clone
Subversion Checkout
CVS CVS Checkout
Import as qmake Project (Limited Fu
Import Existing Project
Choose...
Cancel

Imports existing projects that do not use qmake, CMake or Autotools. This allows you to use Qt Creator as a code editor.

Platform independent

- Under **Location** select the Godot root folder.

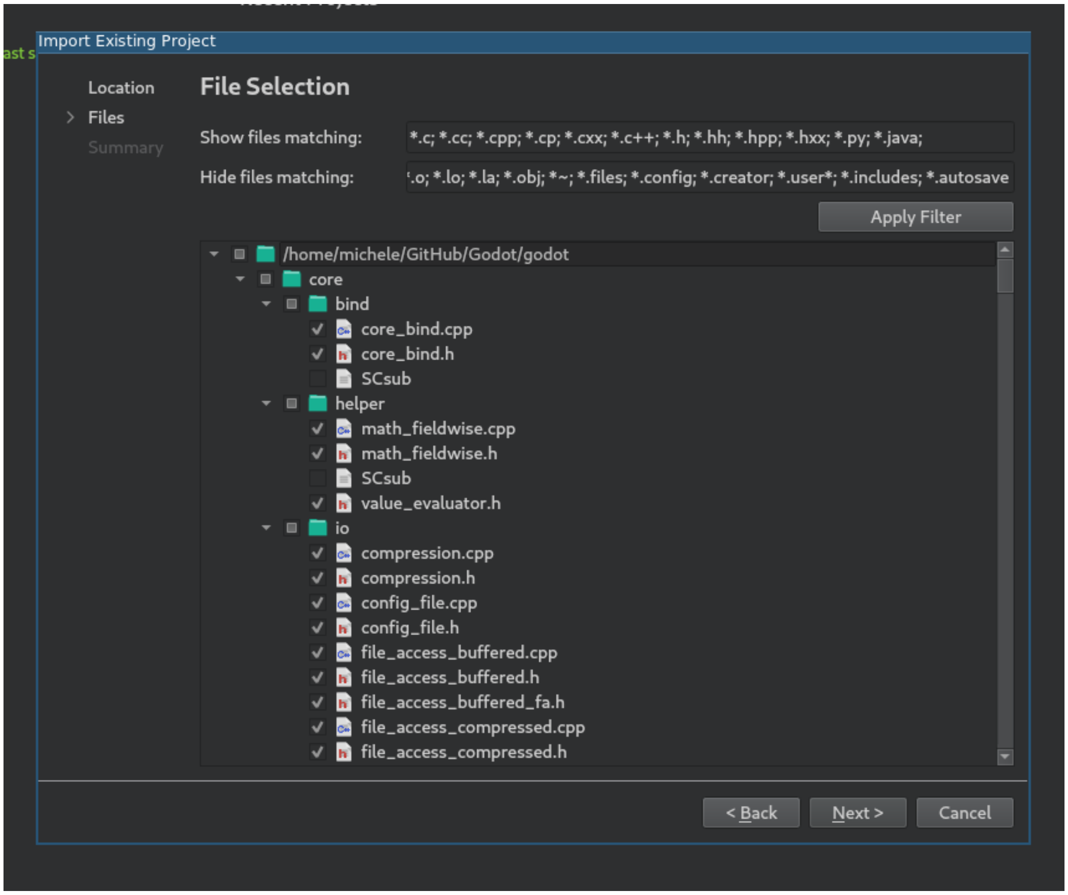

Editor
Crea
htTe
Crea
Import Existing Project
Location
Files
Summary
Next >
Cancel
Browse...

Project Name and Location

Project name: Godot

Location: ichele/GitHub/Godot/godot

- Next, you can choose which folders and files will be visible to the project. While C/C++ files are added automatically, other extensions can be potentially useful: `*.glsl` for shader files, `*.py` for buildsystem files, `*.java` for Android platform development, `*.mm` for macOS platform development.


Import Existing Project
Location
Files
Summary
Apply Filter
< Back
Next >
Cancel

File Selection

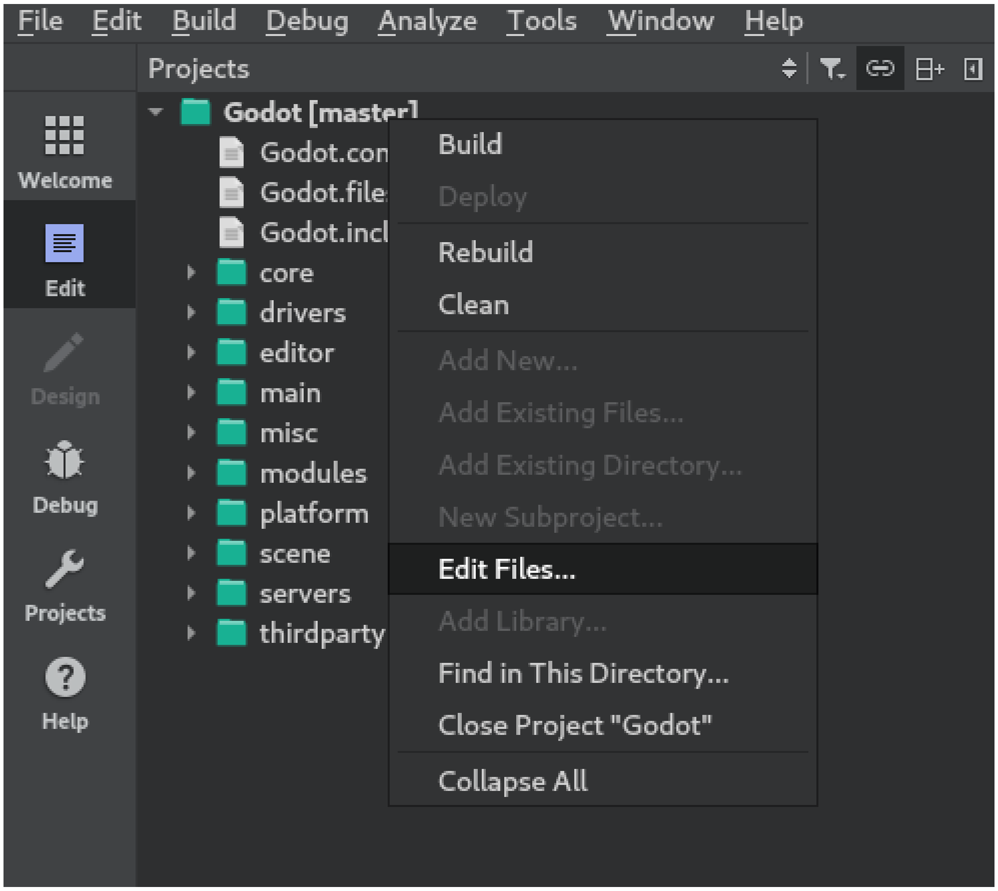
Show files matching: *.c; *.cc; *.cpp; *.cp; *.cxx; *.c++; *.h; *.hh; *.hpp; *.hxx; *.py; *.java;

Hide files matching: '.o; *.lo; *.la; *.obj; *~; *.files; *.config; *.creator; *.user*; *.includes; *.autosave

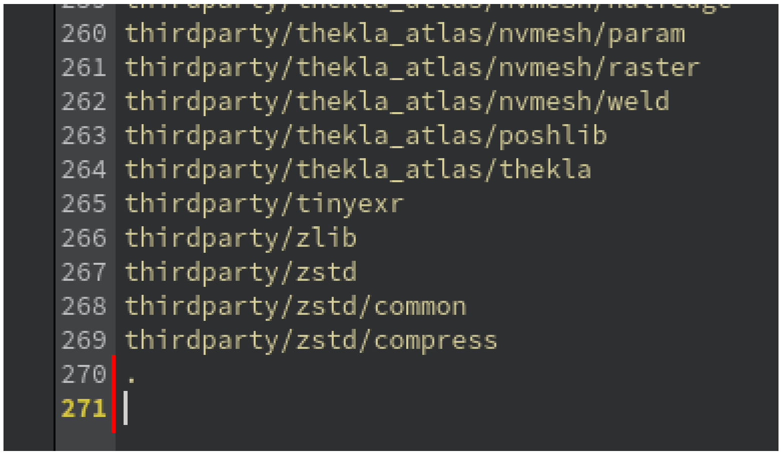
- /home/michele/GitHub/Godot/godot
  - core
    - bind
      - ☑ core_bind.cpp
      - ☑ core_bind.h
      - ☐ SCsub
    - helper
      - ☑ math_fieldwise.cpp
      - ☑ math_fieldwise.h
      - ☐ SCsub
      - ☑ value_evaluator.h
    - io
      - ☑ compression.cpp
      - ☑ compression.h
      - ☑ config_file.cpp
      - ☑ config_file.h
      - ☑ file_access_buffered.cpp
      - ☑ file_access_buffered.h
      - ☑ file_access_buffered_fa.h
      - ☑ file_access_compressed.cpp
      - ☑ file_access_compressed.h

Note

You can change this configuration later by right-clicking on your project and selecting the **Edit Files...** option.

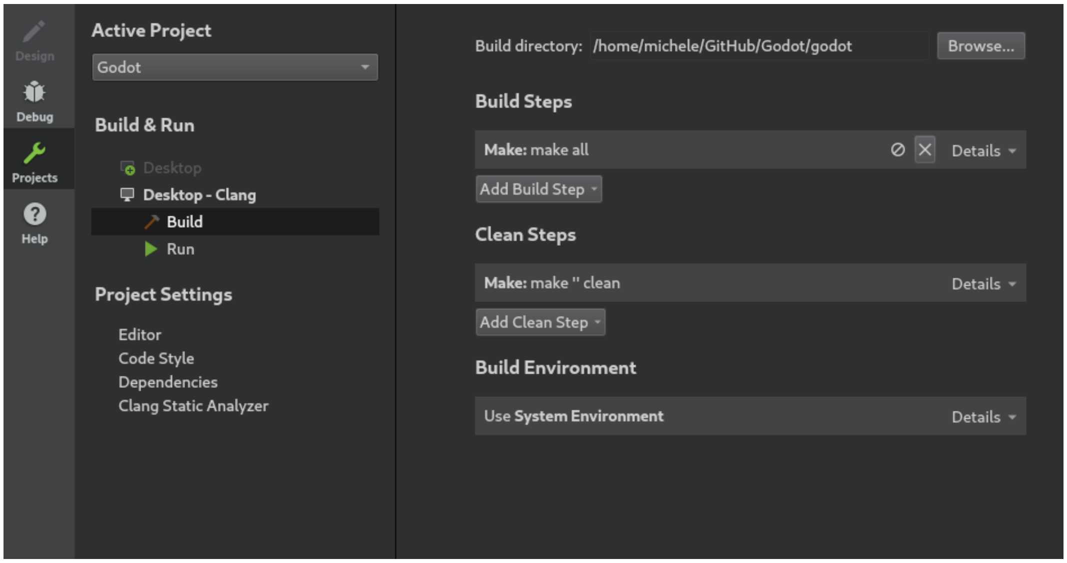

File
Edit
Build
Debug
Analyze
Tools
Window
Help
Projects
Welcome
Edit
Design
Debug
Projects
Help
Godot [master]
Godot.con
Godot.file
Godot.incl
core
drivers
editor
main
misc
modules
platform
scene
servers
thirdparty
Build
Deploy
Rebuild
Clean
Add New...
Add Existing Files...
Add Existing Directory...
New Subproject...
Edit Files...
Add Library...
Find in This Directory...
Close Project "Godot"
Collapse All

- Finish the import.
- Open the `project_name.includes` file and add a line containing `.` to it to correctly enable the code completion.



```
thirdparty/thekla_atlas/nvmesh/param
thirdparty/thekla_atlas/nvmesh/raster
thirdparty/thekla_atlas/nvmesh/weld
thirdparty/thekla_atlas/poshlib
thirdparty/thekla_atlas/thekla
thirdparty/tinyexr
thirdparty/zlib
thirdparty/zstd
thirdparty/zstd/common
thirdparty/zstd/compress
.
```

- From the left-side menu select **Projects** and open the **Build** tab.
- Delete the predefined `make` build step.


Design
Debug
Projects
Help
Active Project
Godot
Build & Run
Desktop
Desktop - Clang
Build
Run
Project Settings
Editor
Code Style
Dependencies
Clang Static Analyzer
Browse...
Build Steps
Details
Add Build Step
Clean Steps
Details
Add Clean Step
Build Environment
Details

Build directory: /home/michele/GitHub/Godot/godot

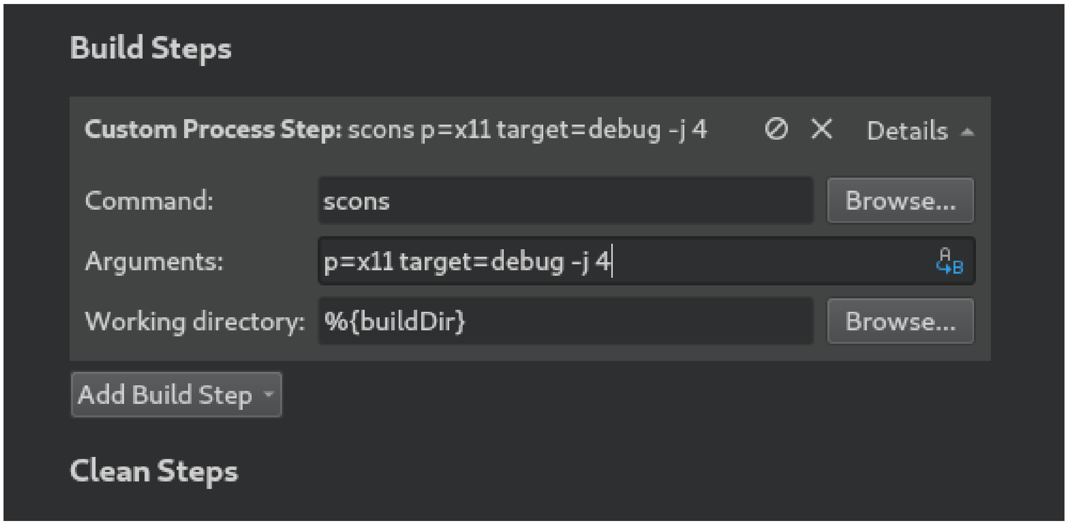
Make: make all

Make: make '' clean

Use **System Environment**

- Click **Add Build Step > Custom Process Step** to add a new build step with the following settings:

Command	**scons**
Arguments	See [Introduction to the buildsystem]() for a full list of arguments.


Build Steps

Custom Process Step: scons p=x11 target=debug -j 4

Command: scons

Arguments: p=x11 target=debug -j 4

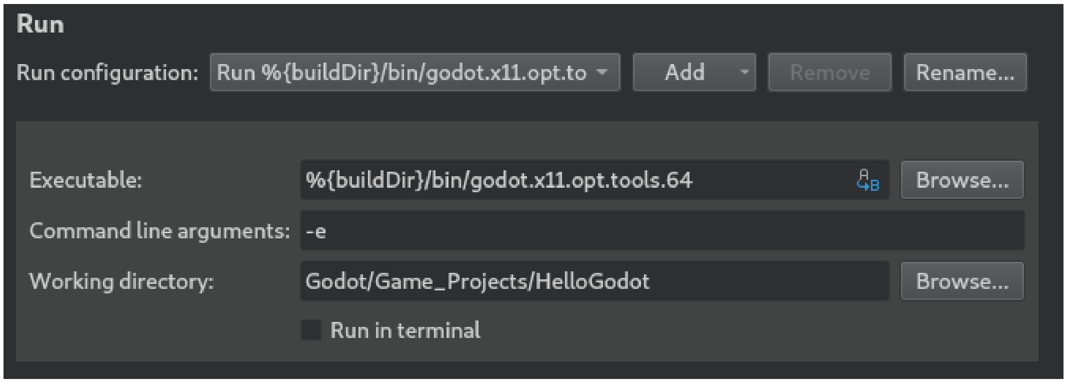
Working directory: %{buildDir}

Note

If the build fails with `Could not start process "scons"`, it can mean that `scons` is not in your `PATH` environment variable. In this case, you'll have to specify the full path to the SCons binary.

Debugging the project

- From the left-side menu select **Projects** and open the **Run** tab.
- Under **Executable** specify the path to your executable located in the `<Godot root directory>/bin` folder. The name depends on your build configuration, e.g. `godot.linuxbsd.editor.dev.x86_64` for 64-bit LinuxBSD platform with `platform=editor` and `dev_build=yes`. You can use `%{buildDir}` to reference the project root, e.g: `%{buildDir}/bin/godot.linuxbsd.editor.dev.x86_64`.
- If you want to run a specific project, specify its root folder under **Working directory**.
- If you want to run the editor, add `-e` to the **Command line arguments** field.


Run
Add
Remove
Rename...
Browse...
Browse...

Run configuration: Run %{buildDir}/bin/godot.x11.opt.to

Executable: %{buildDir}/bin/godot.x11.opt.tools.64

Command line arguments: -e

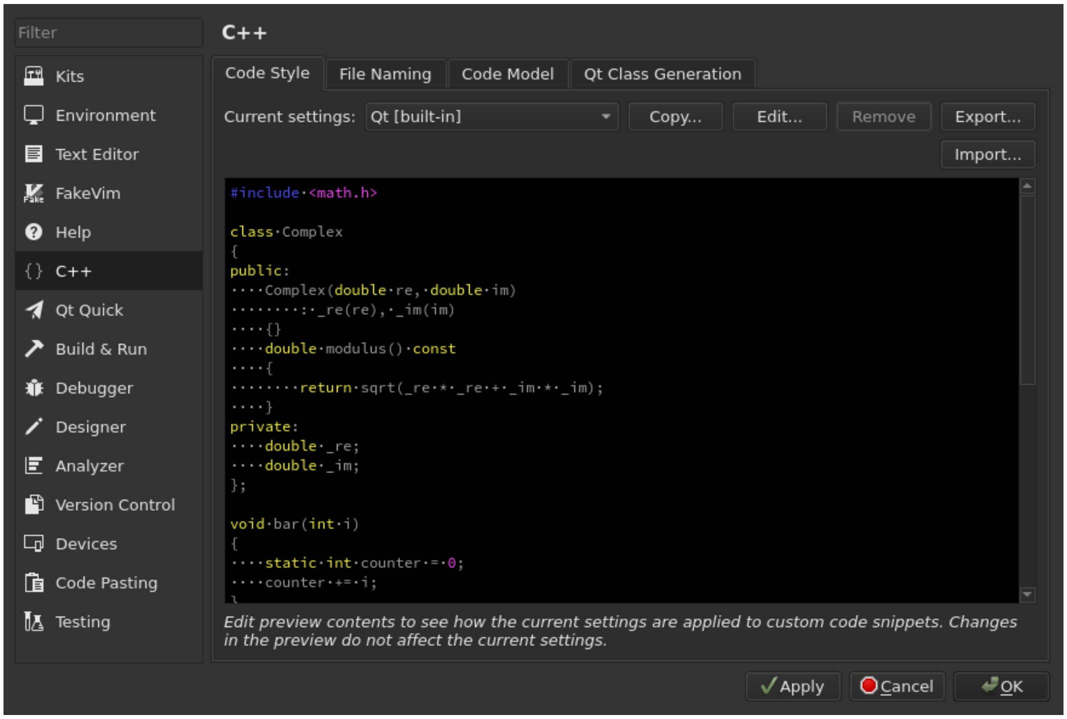
Working directory: Godot/Game_Projects/HelloGodot

☐ Run in terminal

To learn more about command line arguments, refer to the command line tutorial.

Code style configuration

Developers must follow the project's code style and the IDE should help them follow it. By default, Qt Creator uses spaces for indentation which doesn't match the Godot code style guidelines. You can change this behavior by changing the **Code Style** in **Tools > Options > C++**.

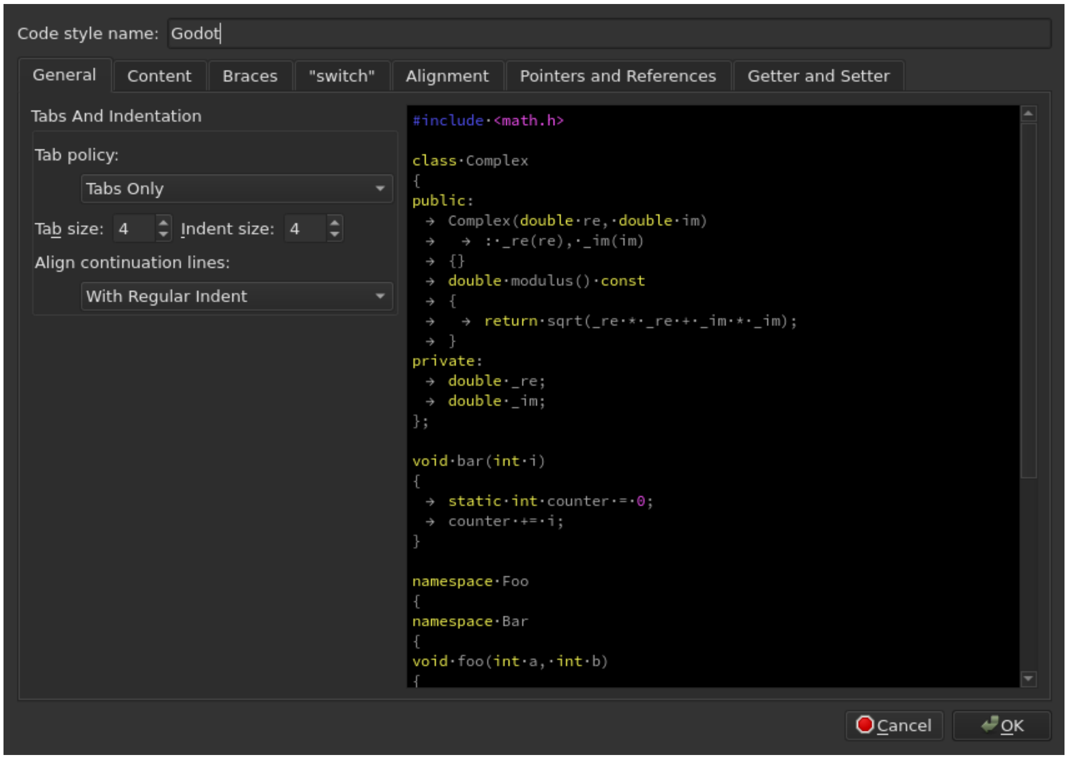

Filter
Kits
Environment
Text Editor
FakeVim
Help
C++
Qt Quick
Build & Run
Debugger
Designer
Analyzer
Version Control
Devices
Code Pasting
Testing
C++
Code Style
File Naming
Code Model
Qt Class Generation
Copy...
Edit...
Remove
Export...
Import...
Apply
Cancel
OK

Current settings: Qt [built-in]

```
#include·<math.h>

class·Complex
{
public:
····Complex(double·re,·double·im)
········:·_re(re),·_im(im)
····{}
····double·modulus()·const
····{
········return·sqrt(_re·*·_re·+·_im·*·_im);
····}
private:
····double·_re;
····double·_im;
};

void·bar(int·i)
{
····static·int·counter·=·0;
····counter·+=·i;
```

Edit preview contents to see how the current settings are applied to custom code snippets. Changes in the preview do not affect the current settings.

Click on **Edit** to change the current settings, then click on **Copy Built-in Code Style** button to set a new code style. Set a name for it (e.g. Godot) and change the Tab policy to be **Tabs Only**.



Code style name: Godot

Tabs And Indentation

Tab policy: Tabs Only

Tab size: 4 Indent size: 4

Align continuation lines: With Regular Indent

```
#include·<math.h>

class·Complex
{
public:
 →  Complex(double·re,·double·im)
 →    →  :·_re(re),·_im(im)
 →  {}
 →  double·modulus()·const
 →  {
 →    →  return·sqrt(_re·*·_re·+·_im·*·_im);
 →  }
private:
 →  double·_re;
 →  double·_im;
};

void·bar(int·i)
{
 →  static·int·counter·=·0;
 →  counter·+=·i;
}

namespace·Foo
{
namespace·Bar
{
void·foo(int·a,·int·b)
{
```

General
Content
Braces
"switch"
Alignment
Pointers and References
Getter and Setter
Cancel
OK

If you run into any issues, ask for help in one of [Godot's community channels](https://godotengine.org/community) [https://godotengine.org/community].

Rider

Rider [https://www.jetbrains.com/rider/] is a commercial JetBrains [https://www.jetbrains.com/] IDE for C# and C++ that uses the same solution system as Visual Studio.

Importing the project

Tip

If you already use Visual Studio as your main IDE, you can use the same solution file in Rider. Rider and Visual Studio use the same solution format, so you can switch between the two IDEs without rebuilding the solution file. Debug configurations need to be changed when going from one IDE to another.

Rider requires a solution file to work on a project. While Godot does not come with the solution file, it can be generated using SCons.

- Navigate to the Godot root folder and open a Command Prompt or PowerShell window.
- Copy, paste and run the next command to generate the solution.

```
scons platform=windows vsproj=yes dev_build=yes
```

The `vsproj` parameter signals that you want Visual Studio solution generated. The `dev_build` parameter makes sure the debug symbols are included, allowing to e.g. step through code using breakpoints.

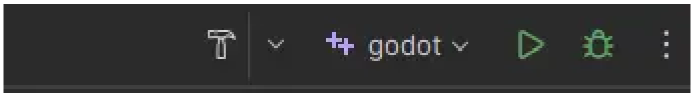
- If you have Rider setup as your main IDE for .sln, you can now open the project by double-clicking on the `godot.sln` in the project root or by using the **Open** option inside of Rider.

Note

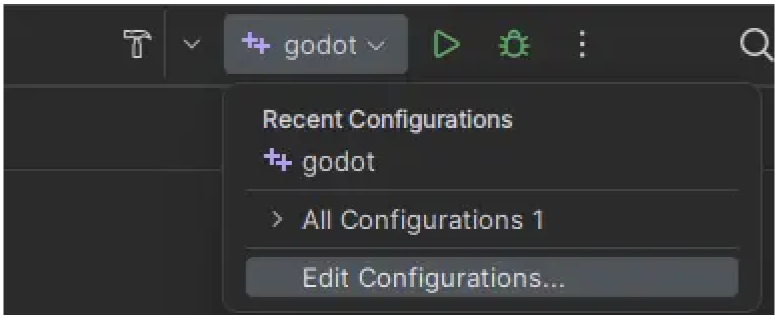
Rider could fail to build the first time you open the project. If that is the case, you can close Rider and open the solution in Visual Studio, build it, close Visual Studio, and open the solution in Rider again. You are now able to build the project from Rider.

Compiling and debugging the project

Rider comes with a built-in debugger that can be used to debug the Godot project. You can launch the debugger by pressing the **Debug** icon at the top of the screen, this only works for the Project manager, if you want to debug the editor, you need to configure the debugger first.



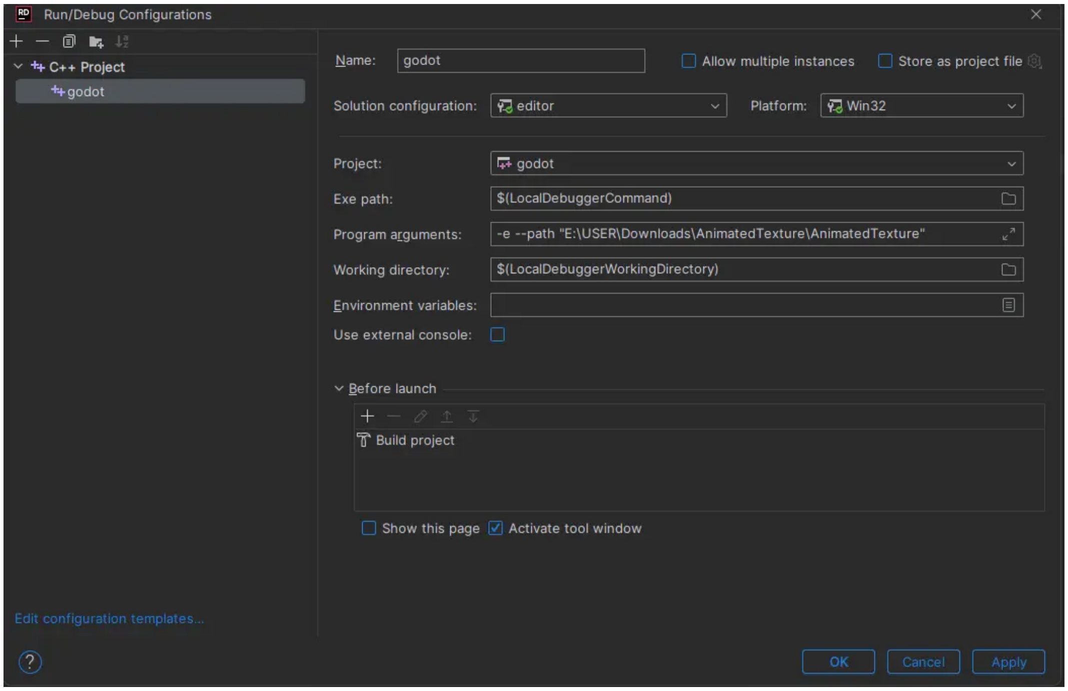
- Click on the **Godot > Edit Configurations** option at the top of the screen.


godot
Recent Configurations
godot
All Configurations 1
Edit Configurations...

- Delete what is under **Program arguments** and paste the following line:

```
-e --path <path to the Godot project>
```

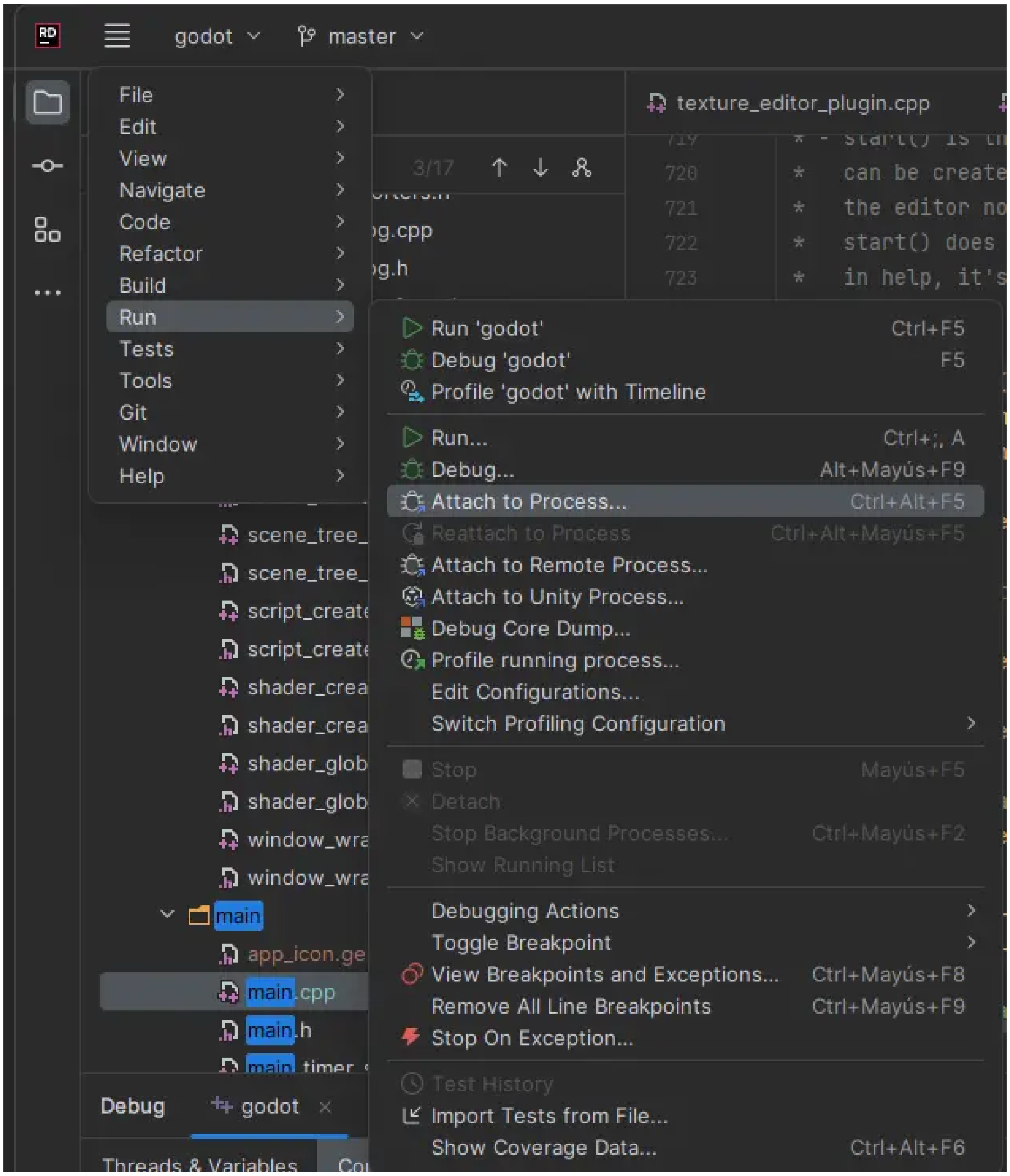
This will tell the executable to debug the specified project without using the project manager. Use the root path to the project folder, not `project.godot` file path.



Name: godot
☐ Allow multiple instances
☐ Store as project file
Solution configuration: editor
Platform: Win32
Project: godot
Exe path: $(LocalDebuggerCommand)
Program arguments: -e --path "E:\USER\Downloads\AnimatedTexture\AnimatedTexture"
Working directory: $(LocalDebuggerWorkingDirectory)
Environment variables:
Use external console: ☐
Before launch
Build project
☐ Show this page
☑ Activate tool window

- Finally click on apply and ok to save the changes.
- You can now launch the debugger by pressing the **Debug** icon at the top of the screen and it will open the project to debug.

Alternatively you can use **Run > Attach to Process** to attach the debugger to a running Godot instance.


godot
master
File
Edit
View
Navigate
Code
Refactor
Build
Run
Tests
Tools
Git
Window
Help
texture_editor_plugin.cpp
3/17
Run 'godot' Ctrl+F5
Debug 'godot' F5
Profile 'godot' with Timeline
Run... Ctrl+;, A
Debug... Alt+Mayús+F9
Attach to Process... Ctrl+Alt+F5
Reattach to Process Ctrl+Alt+Mayús+F5
Attach to Remote Process...
Attach to Unity Process...
Debug Core Dump...
Profile running process...
Edit Configurations...
Switch Profiling Configuration
Stop Mayús+F5
Detach
Stop Background Processes... Ctrl+Mayús+F2
Show Running List
Debugging Actions
Toggle Breakpoint
View Breakpoints and Exceptions... Ctrl+Mayús+F8
Remove All Line Breakpoints Ctrl+Mayús+F9
Stop On Exception...
Test History
Import Tests from File...
Show Coverage Data... Ctrl+Alt+F6
scene_tree_
scene_tree_
script_create
script_create
shader_crea
shader_crea
shader_glob
shader_glob
window_wra
window_wra
main
app_icon.ge
main.cpp
main.h
main_timer
Debug
godot
Threads & Variables

Visual Studio

Visual Studio Community [https://visualstudio.microsoft.com] is a Windows-only IDE by Microsoft [https://microsoft.com] that's free for individual use or non-commercial use within organizations. It has many useful features, such as memory view, performance view, source control and more.

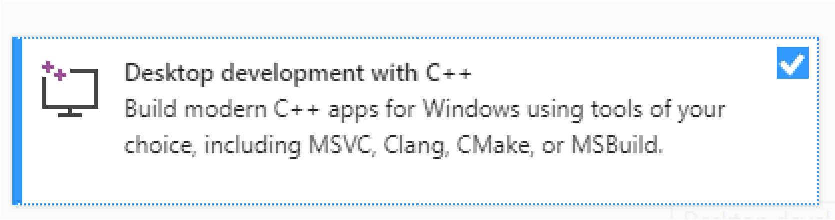
Importing the project

Visual Studio requires a solution file to work on a project. While Godot does not come with the solution file, it can be generated using SCons.

- Navigate to the Godot root folder and open a Command Prompt or PowerShell window.
- Run `scons platform=windows vsproj=yes dev_build=yes` to generate the solution with debug symbols.
  The `vsproj` parameter signals that you want Visual Studio solution generated.
  The `dev_build` parameter makes sure the debug symbols are included, allowing to e.g. step through code using breakpoints.
- You can now open the project by double-clicking on the `godot.sln` in the project root or by using the **Open a project or solution** option inside of the Visual Studio.
- Use the **Build** top menu to build the project.

Warning

Visual Studio must be configured with the C++ package. It can be selected in the installer:



Desktop development with C++

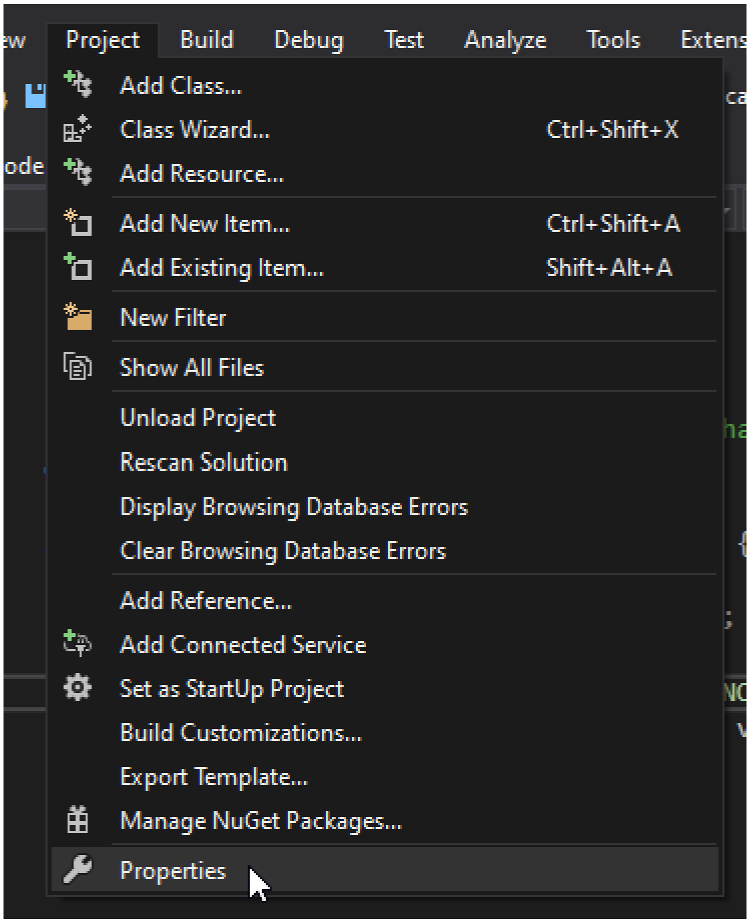
Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

Debugging the project

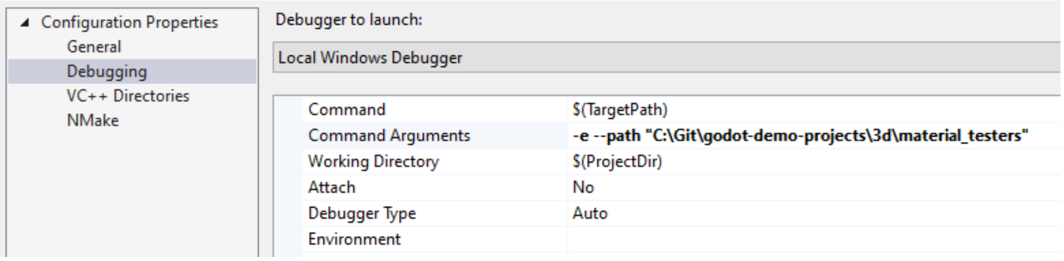
Visual Studio features a powerful debugger. This allows the user to examine Godot's source code, stop at specific points in the code, inspect the current execution context, and make live changes to the codebase.

You can launch the project with the debugger attached using the **Debug > Start Debugging** option from the top menu. However, unless you want to debug the Project Manager specifically, you'd need to configure debugging options first. This is due to the fact that when the Godot Project Manager opens a project, the initial process is terminated and the debugger gets detached.

- To configure the launch options to use with the debugger use **Project > Properties** from the top menu:


ew
Project
Build
Debug
Test
Analyze
Tools
Extens
Add Class...
Class Wizard... Ctrl+Shift+X
Add Resource...
Add New Item... Ctrl+Shift+A
Add Existing Item... Shift+Alt+A
New Filter
Show All Files
Unload Project
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Add Reference...
Add Connected Service
Set as StartUp Project
Build Customizations...
Export Template...
Manage NuGet Packages...
Properties

- Open the **Debugging** section and under **Command Arguments** add two new arguments: the `-e` flag opens the editor instead of the Project Manager, and the `--path` argument tells the executable to open the specified project (must be provided as an *absolute* path to the project root, not the `project.godot` file; if the path contains spaces be sure to pass it inside double quotation marks).


Configuration Properties
General
Debugging
VC++ Directories
NMake

Debugger to launch:

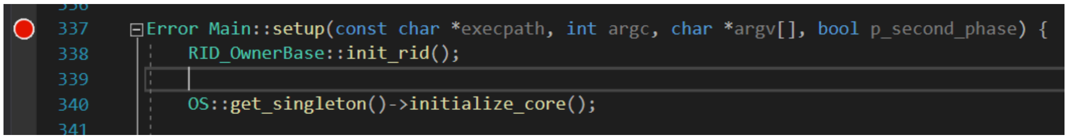
Local Windows Debugger

Command	$(TargetPath)
Command Arguments	**-e --path "C:\Git\godot-demo-projects\3d\material_testers"**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	

To learn more about command line arguments, refer to the command line tutorial.

Even if you start the project without a debugger attached it can still be connected to the running process using **Debug > Attach to Process...** menu.

To check that everything is working, put a breakpoint in `main.cpp` and press `F5` to start debugging.



```
Error Main::setup(const char *execpath, int argc, char *argv[], bool p_second_phase) {
    RID_OwnerBase::init_rid();

    OS::get_singleton()->initialize_core();
```

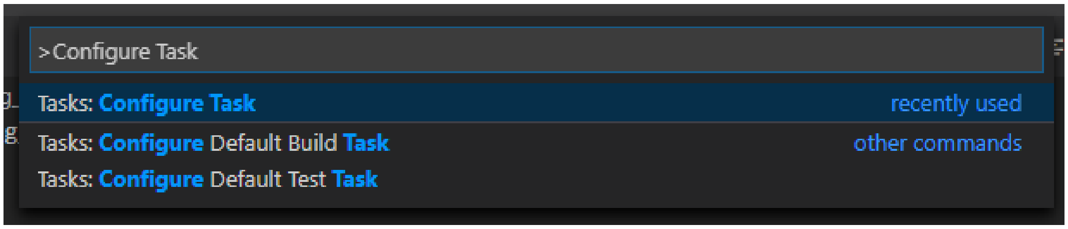
If you run into any issues, ask for help in one of Godot's community channels [https://godotengine.org/community].

Visual Studio Code

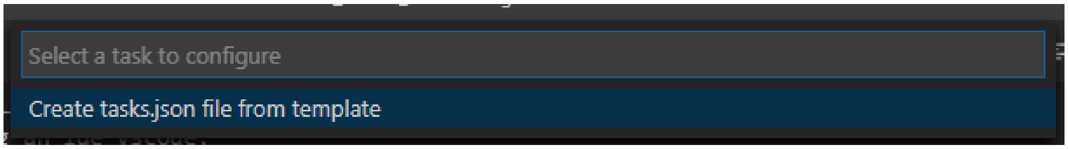
Visual Studio Code [https://code.visualstudio.com] is a free cross-platform code editor by Microsoft [https://microsoft.com] (not to be confused with Visual Studio).

Importing the project

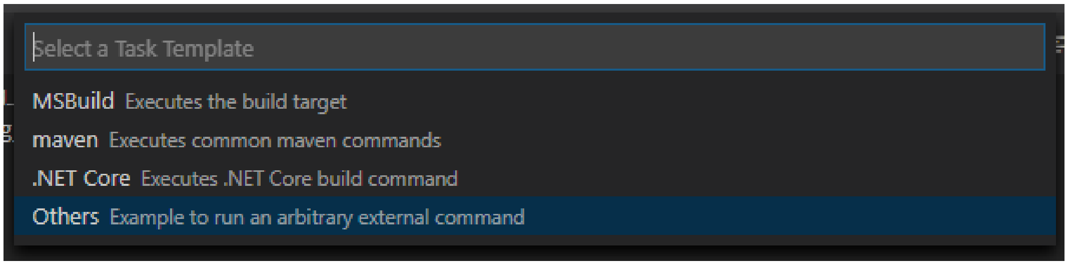
- Make sure the C/C++ extension is installed. You can find instructions in the official documentation [https://code.visualstudio.com/docs/languages/cpp]. Alternatively, clangd [https://open-vsx.org/extension/llvm-vs-code-extensions/vscode-clangd] can be used instead.
- When using the clangd extension, run `scons compiledb=yes`.
- From the Visual Studio Code's main screen open the Godot root folder with **File > Open Folder...**.
- Press `Ctrl + Shift + P` to open the command prompt window and enter *Configure Task*.


>Configure Task
Tasks: Configure Task
recently used
Tasks: Configure Default Build Task
other commands
Tasks: Configure Default Test Task

- Select the **Create tasks.json file from template** option.


Select a task to configure
Create tasks.json file from template

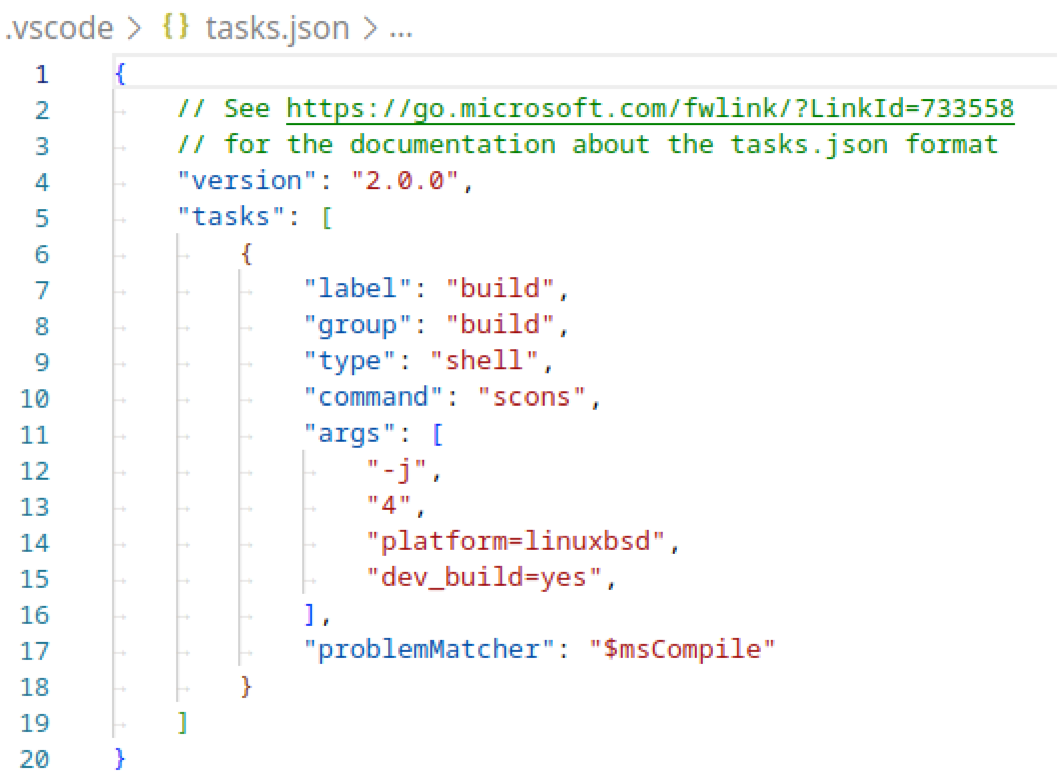
- Then select **Others**.


Select a Task Template
MSBuild Executes the build target
maven Executes common maven commands
.NET Core Executes .NET Core build command
Others Example to run an arbitrary external command

- If there is no such option as **Create tasks.json file from template** available, either delete the file if it already exists in your folder or create a `.vscode/tasks.json` file manually. See [Tasks in Visual Studio Code](https://code.visualstudio.com/docs/editor/tasks#_custom-tasks) [https://code.visualstudio.com/docs/editor/tasks#_custom-tasks] for more details on tasks.

- Within the `tasks.json` file find the `"tasks"` array and add a new section to it:

```
{
  "label": "build",
  "group": "build",
  "type": "shell",
  "command": "scons",
  "args": [
    // enable for debugging with breakpoints
    "dev_build=yes",
  ],
  "problemMatcher": "$msCompile"
}
```



```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "group": "build",
            "type": "shell",
            "command": "scons",
            "args": [
                "-j",
                "4",
                "platform=linuxbsd",
                "dev_build=yes",
            ],
            "problemMatcher": "$msCompile"
        }
    ]
}
```

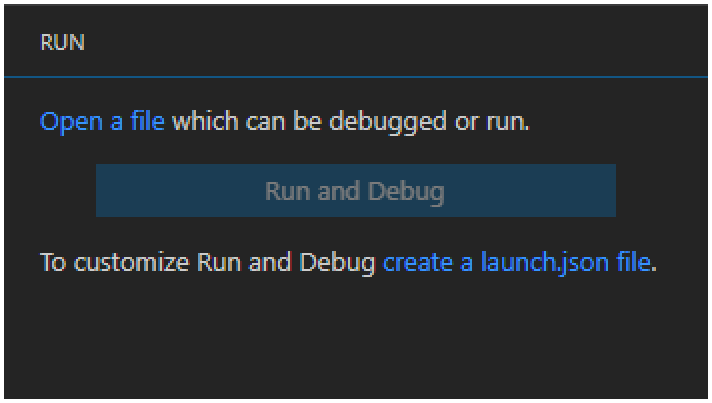
An example of a filled out `tasks.json`.

Arguments can be different based on your own setup and needs. See [Introduction to the buildsystem]() for a full list of arguments.

Debugging the project

To run and debug the project you need to create a new configuration in the `launch.json` file.

- Press `Ctrl + Shift + D` to open the Run panel.
- If `launch.json` file is missing you will be prompted to create a new one.



RUN

Open a file which can be debugged or run.

To customize Run and Debug create a launch.json file.

- Select **C++ (GDB/LLDB)**. There may be another platform specific option here. If selected, adjust the configuration example provided accordingly.
- Within the `launch.json` file find the `"configurations"` array and add a new section to it:

LinuxBSD

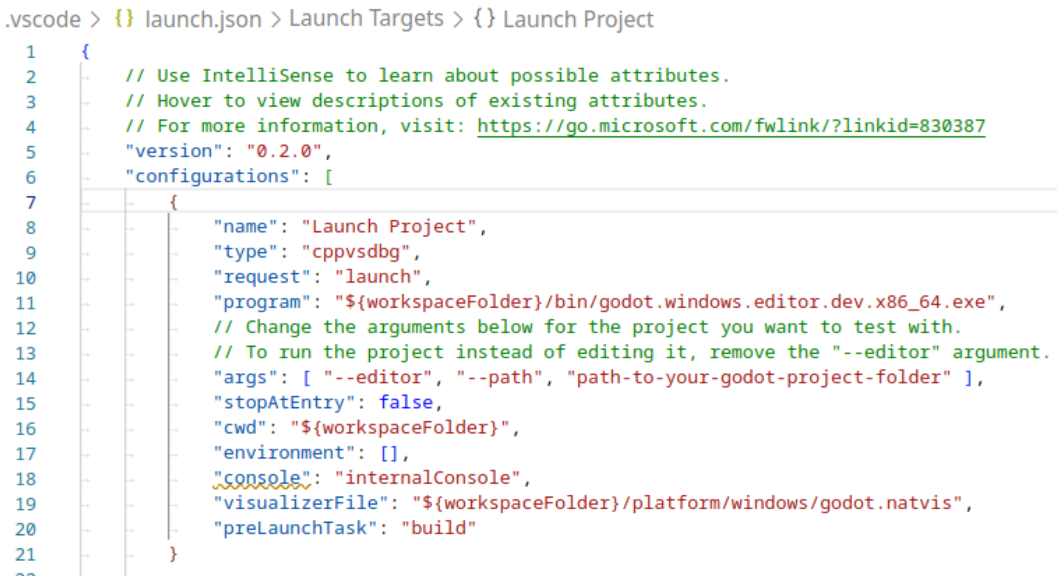
```
{
  "name": "Launch Project",
  "type": "lldb",
  "request": "launch",
  // Change to godot.linuxbsd.editor.dev.x86_64.llvm for llvm-based builds.
  "program": "${workspaceFolder}/bin/godot.linuxbsd.editor.dev.x86_64",
  // Change the arguments below for the project you want to test with.
  // To run the project instead of editing it, remove the "--editor" argument.
  "args": [ "--editor", "--path", "path-to-your-godot-project-folder" ],
  "stopAtEntry": false,
  "cwd": "${workspaceFolder}",
  "environment": [],
  "externalConsole": false,
  "preLaunchTask": "build"
}
```

LinuxBSD_gdb

```
{
  "name": "Launch Project",
  "type": "cppdbg",
  "request": "launch",
  // Change to godot.linuxbsd.editor.dev.x86_64.llvm for llvm-based builds.
  "program": "${workspaceFolder}/bin/godot.linuxbsd.editor.dev.x86_64",
  // Change the arguments below for the project you want to test with.
  // To run the project instead of editing it, remove the "--editor" argument.
  "args": [ "--editor", "--path", "path-to-your-godot-project-folder" ],
  "stopAtEntry": false,
  "cwd": "${workspaceFolder}",
  "environment": [],
  "externalConsole": false,
  "setupCommands":
  [
    {
      "description": "Enable pretty-printing for gdb",
      "text": "-enable-pretty-printing",
      "ignoreFailures": true
    },
    {
        "description": "Load custom pretty-printers for Godot types.",
        "text": "source ${workspaceRoot}/misc/utility/godot_gdb_pretty_print.py"
    }
  ],
  "preLaunchTask": "build"
}
```

Windows

```
{
  "name": "Launch Project",
  "type": "cppvsdbg",
  "request": "launch",
  "program": "${workspaceFolder}/bin/godot.windows.editor.dev.x86_64.exe",
  // Change the arguments below for the project you want to test with.
  // To run the project instead of editing it, remove the "--editor" argument.
  "args": [ "--editor", "--path", "path-to-your-godot-project-folder" ],
  "stopAtEntry": false,
  "cwd": "${workspaceFolder}",
  "environment": [],
  "console": "internalConsole",
  "visualizerFile": "${workspaceFolder}/platform/windows/godot.natvis",
  "preLaunchTask": "build"
}
```


.vscode > {} launch.json > Launch Targets > {} Launch Project

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Launch Project",
            "type": "cppvsdbg",
            "request": "launch",
            "program": "${workspaceFolder}/bin/godot.windows.editor.dev.x86_64.exe",
            // Change the arguments below for the project you want to test with.
            // To run the project instead of editing it, remove the "--editor" argument.
            "args": [ "--editor", "--path", "path-to-your-godot-project-folder" ],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "console": "internalConsole",
            "visualizerFile": "${workspaceFolder}/platform/windows/godot.natvis",
            "preLaunchTask": "build"
        }
```

An example of a filled out `launch.json`.

Note

Due to sporadic performance issues, it is recommended to use LLDB over GDB on Unix-based systems. Make sure that the CodeLLDB extension [https://marketplace.visualstudio.com/items?itemName=vadimcn.vscode-lldb] is installed.

If you encounter issues with lldb, you may consider using gdb (see the LinuxBSD_gdb configuration).

Do note that lldb may work better with LLVM-based builds. See [Compiling for Linux, *BSD]() for further information.

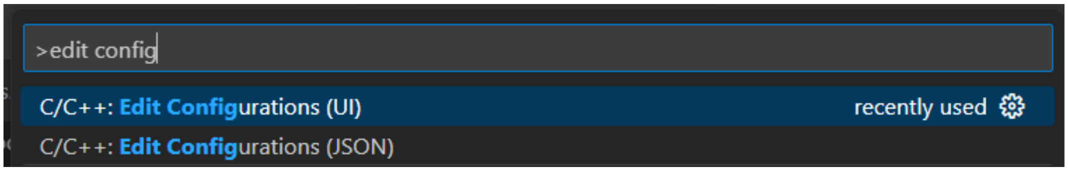
The name under `program` depends on your build configuration, e.g. `godot.linuxbsd.editor.dev.x86_64` for 64-bit LinuxBSD platform with `target=editor` and `dev_build=yes`.

Configuring Intellisense

For the C/C++ extension:

To fix include errors you may be having, you need to configure some settings in the `c_cpp_properties.json` file.

- First, make sure to build the project since some files need to be generated.
- Edit the C/C++ Configuration file either with the UI or with text:



>edit config
C/C++: Edit Configurations (UI)
recently used
C/C++: Edit Configurations (JSON)

- Add an include path for your platform, for example, `${workspaceFolder}/platform/windows`.
- Add defines for the editor `TOOLS_ENABLED`, debug builds `DEBUG_ENABLED`, and tests `TESTS_ENABLED`.
- Make sure the compiler path is configured correctly to the compiler you are using. See [Introduction to the buildsystem]() for further information on your platform.

- The `c_cpp_properties.json` file should look similar to this for Windows:

```
{
  "configurations": [
    {
      "name": "Win32",
      "includePath": [
        "${workspaceFolder}/**",
        "${workspaceFolder}/platform/windows"
      ],
      "defines": [
        "_DEBUG",
        "UNICODE",
        "_UNICODE",
        "TOOLS_ENABLED",
        "DEBUG_ENABLED",
        "TESTS_ENABLED"
      ],
      "windowsSdkVersion": "10.0.22621.0",
      "compilerPath": "C:/Program Files/Microsoft Visual Studio/2022/Community/VC/Tools/MSVC/14.39.33519/bin/Hostx64/x64/cl.exe",
      "cStandard": "c17",
      "cppStandard": "c++17",
      "intelliSenseMode": "windows-msvc-x64"
    }
  ],
  "version": 4
}
```

- Alternatively, you can use the scons argument `compiledb=yes` and set the compile commands setting `compileCommands` to `compile_commands.json`, found in the advanced section of the C/C++ Configuration UI.
  - This argument can be added to your build task in `tasks.json` since it will need to be run whenever files are added or moved.

If you run into any issues, ask for help in one of Godot's community channels [https://godotengine.org/community].

Tip

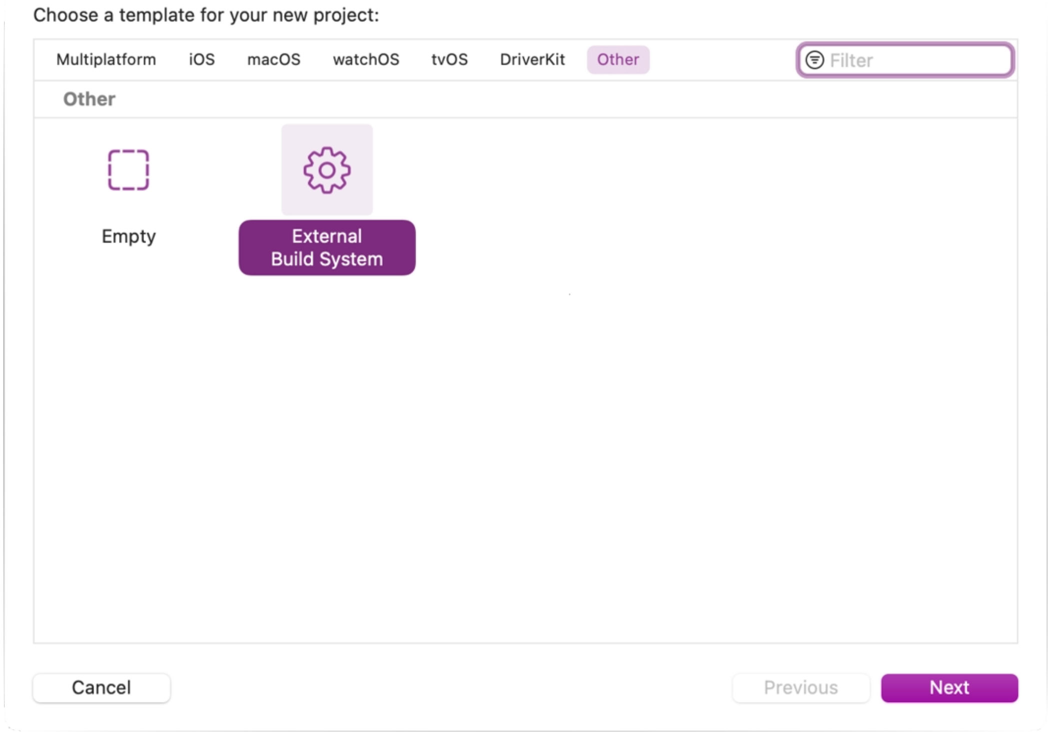
To get linting on class reference XML files, install the vscode-xml extension [https://marketplace.visualstudio.com/items?itemName=redhat.vscode-xml].

Xcode

Xcode [https://developer.apple.com/xcode] is a free macOS-only IDE. You can download it from the Mac App Store.

Importing the project

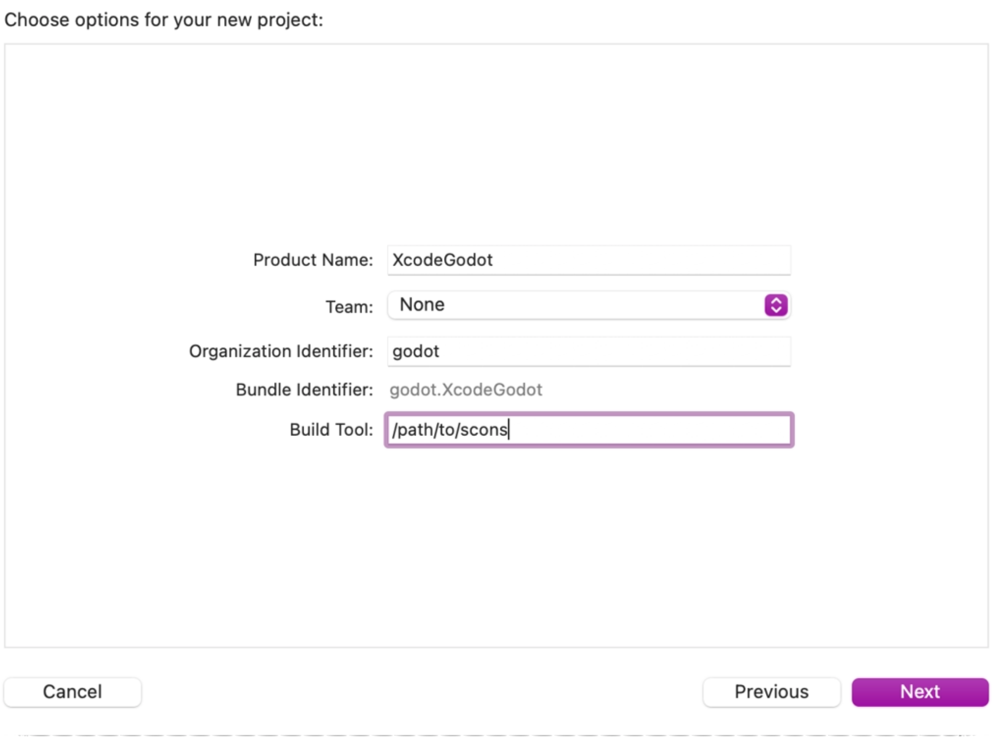
- From Xcode's main screen create a new project using the **Other > External Build System** template.



Choose a template for your new project:

Multiplatform
iOS
macOS
watchOS
tvOS
DriverKit
Other
Filter
Other
Empty
External Build System
Cancel
Previous
Next

- Now choose a name for your project and set the path to scons executable in build tool (to find the path you can type `where scons` in a terminal).



Choose options for your new project:

Product Name: XcodeGodot

Team: None

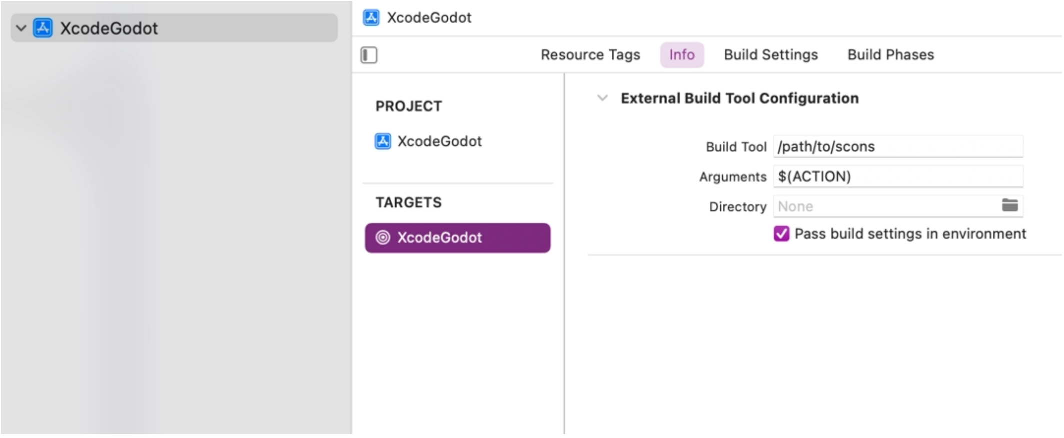
Organization Identifier: godot

Bundle Identifier: godot.XcodeGodot

Build Tool: /path/to/scons

Cancel
Previous
Next

- Open the main target from the **Targets** section and select the **Info** tab.



XcodeGodot
XcodeGodot
Resource Tags
Info
Build Settings
Build Phases
PROJECT
XcodeGodot
TARGETS
XcodeGodot
External Build Tool Configuration

Build Tool: /path/to/scons

Arguments: $(ACTION)

Directory: None

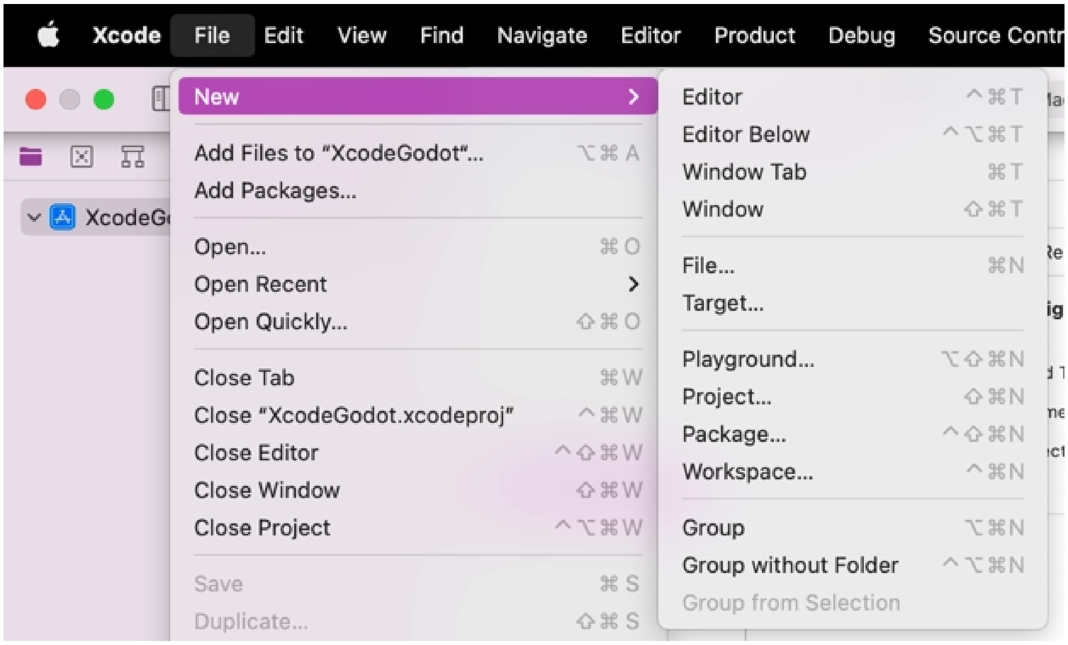
☑ Pass build settings in environment

- Fill out the form with the following settings:

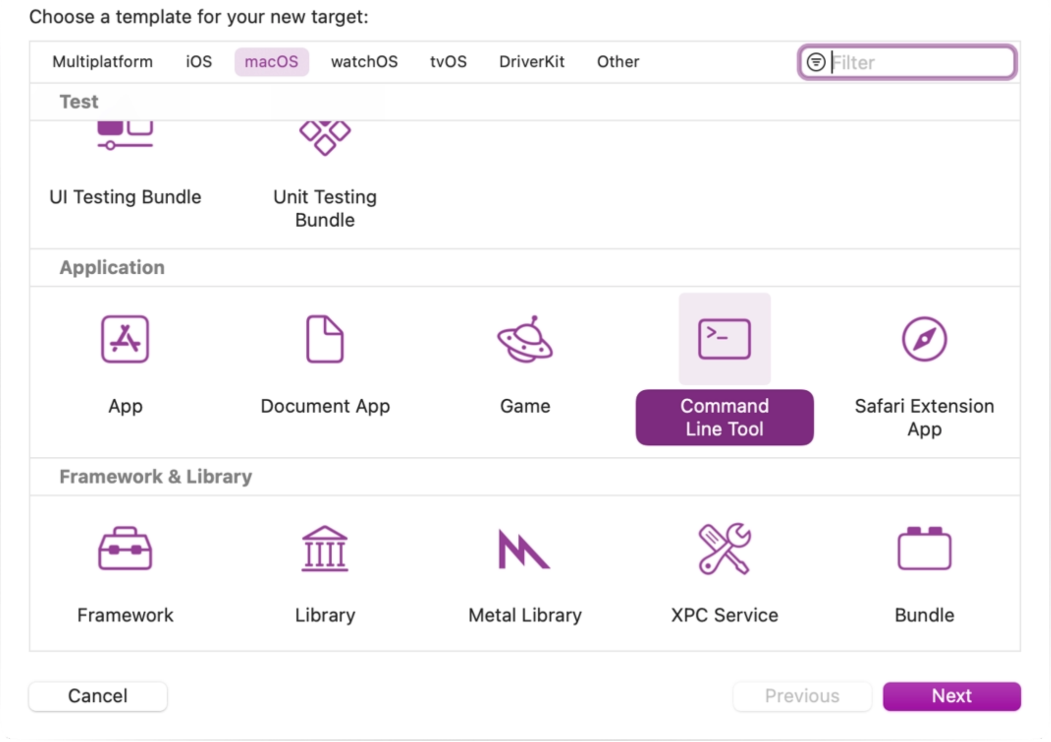
Arguments	See Introduction to the buildsystem for a full list of arguments.

Directory	A full path to the Godot root folder

- Add a Command Line Tool target which will be used for indexing the project by choosing **File > New > Target...**.


Xcode
File
Edit
View
Find
Navigate
Editor
Product
Debug
Source Contr
New
Add Files to "XcodeGodot"... ⌥⌘A
Add Packages...
Open... ⌘O
Open Recent
Open Quickly... ⇧⌘O
Close Tab ⌘W
Close "XcodeGodot.xcodeproj" ^⌘W
Close Editor ^⇧⌘W
Close Window ⇧⌘W
Close Project ^⌥⌘W
Save ⌘S
Duplicate... ⇧⌘S
Editor ^⌘T
Editor Below ^⌥⌘T
Window Tab ⌘T
Window ⇧⌘T
File... ⌘N
Target...
Playground... ⌥⇧⌘N
Project... ⇧⌘N
Package... ^⇧⌘N
Workspace... ^⌘N
Group ⌥⌘N
Group without Folder ^⌥⌘N
Group from Selection
XcodeG

- Select **macOS > Application > Command Line Tool**.

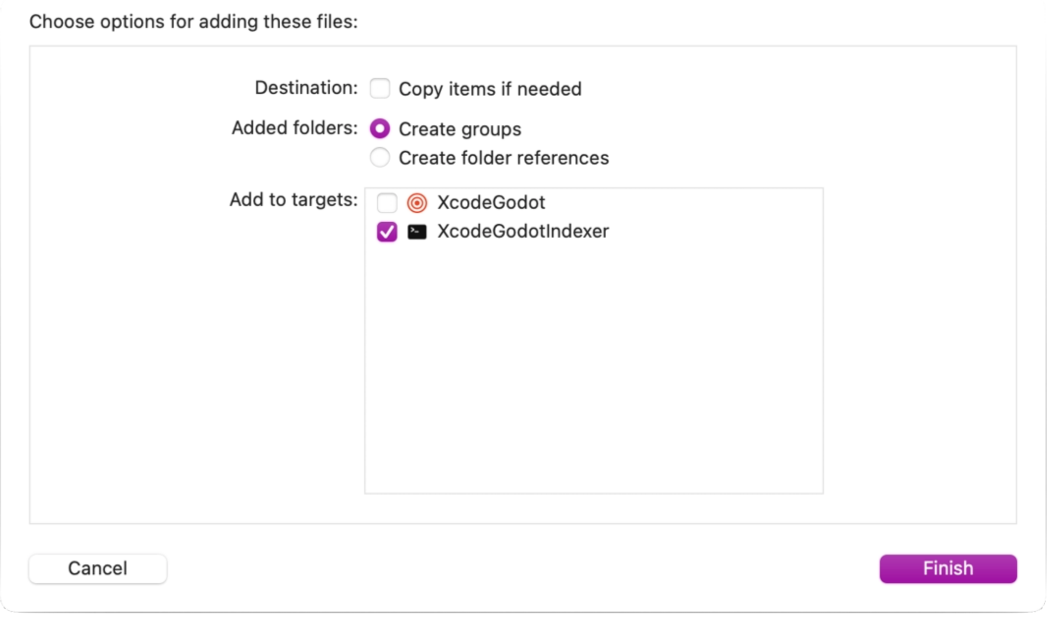

Choose a template for your new target:

Multiplatform
iOS
macOS
watchOS
tvOS
DriverKit
Other
Filter
Test
UI Testing Bundle
Unit Testing Bundle
Application
App
Document App
Game
Command Line Tool
Safari Extension App
Framework & Library
Framework
Library
Metal Library
XPC Service
Bundle
Cancel
Previous
Next

Note

Name it something so you know not to compile with this target (e.g. `GodotXcodeIndex`).

- For this target open the **Build Settings** tab and look for **Header Search Paths**.
- Set **Header Search Paths** to the absolute path to the Godot root folder. You need to include subdirectories as well. To achieve that, add two two asterisks (`**`) to the end of the path, e.g. `/Users/me/repos/godot-source/**`.
- Add the Godot source to the project by dragging and dropping it into the project file browser.
- Select **Create groups** for the **Added folders** option and check *only* your command line indexing target in the **Add to targets** section.


Choose options for adding these files:

Destination: ☐ Copy items if needed

Added folders: ● Create groups
○ Create folder references

Add to targets: ☐ XcodeGodot
☑ XcodeGodotIndexer

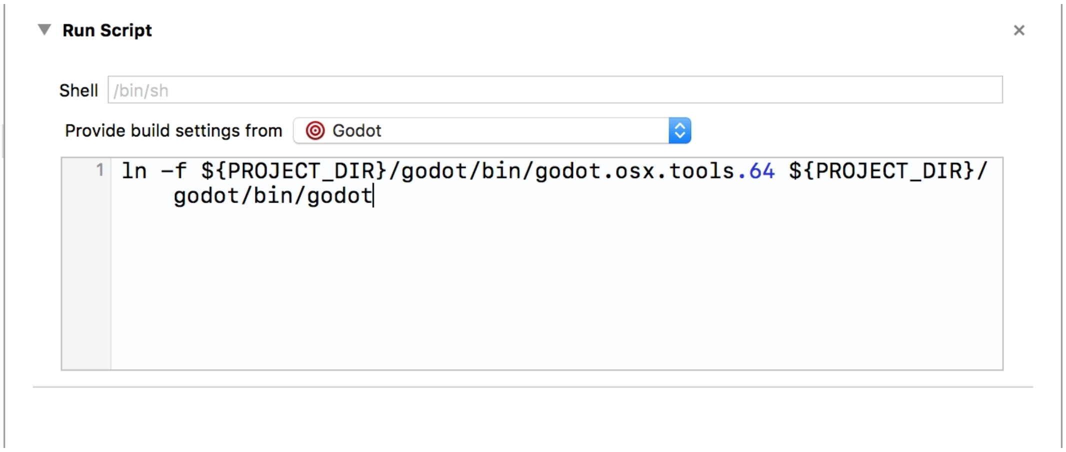
- Xcode will now index the files. This may take a few minutes.
- Once Xcode is done indexing, you should have jump-to-definition, autocompletion, and full syntax highlighting.

Debugging the project

To enable debugging support you need to edit the external build target's build and run schemes.

- Open the scheme editor of the external build target.
- Locate the **Build > Post Actions** section.
- Add a new script run action
- Under **Provide build settings from** select your project. This allows to reference the project directory within the script.
- Create a script that will give the binary a name that Xcode can recognize, e.g.:

```
ln -f ${PROJECT_DIR}/godot/bin/godot.macos.tools.64
${PROJECT_DIR}/godot/bin/godot
```

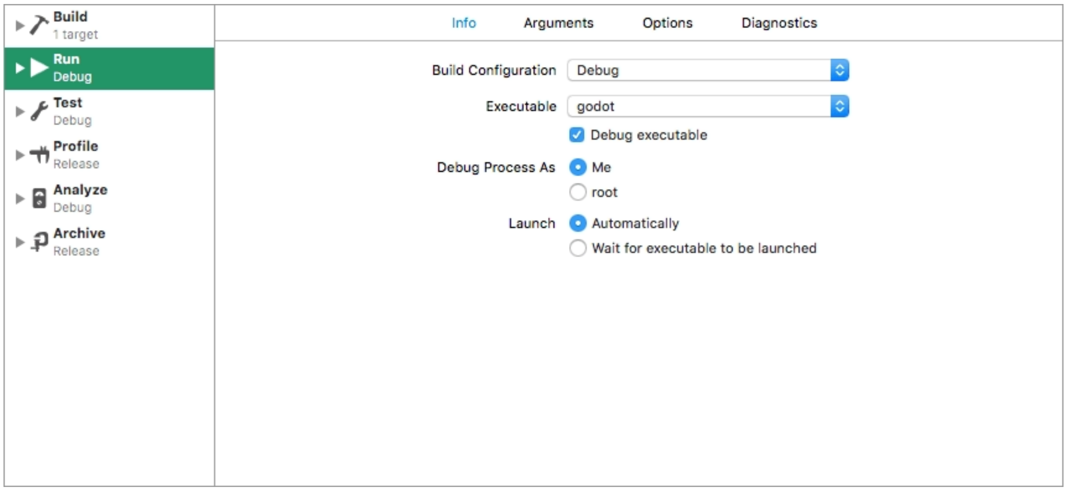


Shell: /bin/sh

Provide build settings from: Godot

```
ln -f ${PROJECT_DIR}/godot/bin/godot.osx.tools.64 ${PROJECT_DIR}/
    godot/bin/godot
```

- Build the external build target.
- Open the scheme editor again and select **Run**.


Build
1 target
Run
Debug
Test
Debug
Profile
Release
Analyze
Debug
Archive
Release
Info
Arguments
Options
Diagnostics

Build Configuration: Debug

Executable: godot

☑ Debug executable

Debug Process As: ● Me ○ root

Launch: ● Automatically ○ Wait for executable to be launched

- Set the **Executable** to the file you linked in your post-build action script.
- Check **Debug executable**.
- You can add two arguments on the **Arguments** tab: the `-e` flag opens the editor instead of the Project Manager, and the `--path` argument tells the executable to open the specified project (must be provided as an *absolute* path to the project root, not the `project.godot` file).

To check that everything is working, put a breakpoint in `platform/macos/godot_main_macos.mm` and run the project.

If you run into any issues, ask for help in one of Godot's community channels [https://godotengine.org/community].

Building from source

Godot prides itself on being very easy to build, by C++ projects' standards. Godot uses the SCons build system, and after the initial setup compiling the engine for your current platform should be as easy as running:

```
scons
```

But you will probably need to use at least some of the available options to configure the build to match your specific needs, be it a custom engine fork, a lightweight build stripped of extra modules, or an executable targeting engine development.

The articles below should help you navigate configuration options available, as well as prerequisites required to compile Godot exactly the way you need.

Basics of building Godot

Let's start with basics, and learn how to get Godot's source code, and then which options to use to compile it regardless of your target platform.

- Getting the source
- Introduction to the buildsystem

Building for target platforms

Below you can find instructions for compiling the engine for your specific target platform. Note that Godot supports cross-compilation, which means you can compile it for a target platform that doesn't match your current platform (say, target Linux while being on

Windows). The guides will try their best to cover all possible situations.

- Compiling for Windows
- Compiling for Linux, *BSD
- Compiling for macOS
- Compiling for Android
- Compiling for iOS
- Cross-compiling for iOS on Linux
- Compiling for the Web

Other compilation targets and options

Some additional universal compilation options require further setup. Namely, while Godot does have C#/.NET support as a part of its main codebase, it does not get compiled by default to reduce the executable size for users who don't need C# for their projects.

Articles below explain how to configure the buildsystem for cases like this, and also cover some optimization techniques.

- Compiling with .NET
- Compiling with PCK encryption key
- Optimizing a build for size

Getting the source

Downloading the Godot source code

Before getting into the SCons build system and compiling Godot, you need to actually download the Godot source code.

The source code is available on GitHub [https://github.com/godotengine/godot] and while you can manually download it via the website, in general you want to do it via the `git` version control system.

If you are compiling in order to make contributions or pull requests, you should follow the instructions from the Pull Request workflow.

If you don't know much about `git` yet, there are a great number of tutorials [https://git-scm.com/book] available on various websites.

In general, you need to install `git` and/or one of the various GUI clients.

Afterwards, to get the latest development version of the Godot source code (the unstable `master` branch), you can use `git clone`.

If you are using the `git` command line client, this is done by entering the following in a terminal:

```
git clone https://github.com/godotengine/godot.git
# You can add the --depth 1 argument to omit the commit
history.
# Faster, but not all Git operations (like blame) will work.
```

For any stable release, visit the release page [https://github.com/godotengine/godot/releases] and click on the link for the release you want. You can then download and extract the source from the download link on the page.

With `git`, you can also clone a stable release by specifying its branch or tag after the `--branch` (or just `-b`) argument:

```
# Clone the continuously maintained stable branch (`3.x` as of
writing).
git clone https://github.com/godotengine/godot.git -b 3.x

# Clone the `3.2.3-stable` tag. This is a fixed revision that
will never change.
git clone https://github.com/godotengine/godot.git -b 3.2.3-
stable
```

There are also generally branches besides `master` for each major version.

After downloading the Godot source code, you can continue to compiling Godot.

Introduction to the buildsystem

Godot is a primarily C++ project and it uses the SCons build system. We love SCons for how maintainable and easy to set up it makes our buildsystem. And thanks to that compiling Godot from source can be as simple as running:

```
scons
```

This produces an editor build for your current platform, operating system, and architecture. You can change what gets built by specifying a target, a platform, and/or an architecture. For example, to build an export template used for running exported games, you can run:

```
scons target=template_release
```

If you plan to debug or develop the engine, then you might want to enable the `dev_build` option to enable dev-only debugging code:

```
scons dev_build=yes
```

Following sections in the article will explain these and other universal options in more detail. But before you can compile Godot, you need to install a few prerequisites. Please refer to the platform documentation to learn more:

- Compiling for Android
- Compiling for iOS
- Compiling for Linux, *BSD
- Compiling for macOS
- Compiling for the Web
- Compiling for Windows

These articles cover in great detail both how to setup your environment to compile Godot on a specific platform, and how to compile for that platform. Please feel free to go back and forth between them and this article to reference platform-specific and universal configuration options.

Using multi-threading

The build process may take a while, depending on how powerful your system is. By default, Godot's SCons setup is configured to use all CPU threads but one (to keep the system responsive during compilation). If you want to adjust how many CPU threads SCons will use, use the `-j <threads>` parameter to specify how many threads will be used for the build.

Example for using 4 threads:

```
scons -j4
```

Platform selection

Godot's build system will begin by detecting the platforms it can build for. If not detected, the platform will simply not appear on the list of available platforms. The build requirements for each platform are described in the rest of this tutorial section.

SCons is invoked by just calling `scons`. If no platform is specified, SCons will detect the target platform automatically based on the host platform. It will then start building for the target platform right away.

To list the available target platforms, use `scons platform=list`:

```
scons platform=list
scons: Reading SConscript files ...
```

```
The following platforms are available:

    android
    javascript
    linuxbsd
    server
    windows

Please run SCons again and select a valid platform: platform=
<string>
```

To build for a platform (for example, `linuxbsd`), run with the `platform=` (or `p=` to make it short) argument:

```
scons platform=linuxbsd
```

Resulting binary

The resulting binaries will be placed in the `bin/` subdirectory, generally with this naming convention:

```
godot.<platform>.<target>[.dev][.double].<arch>[.
<extra_suffix>][.<ext>]
```

For the previous build attempt, the result would look like this:

```
ls bin
bin/godot.linuxbsd.editor.x86_64
```

This means that the binary is for Linux *or* *BSD (*not* both), is not optimized, has the whole editor compiled in, and is meant for 64 bits.

A Windows binary with the same configuration will look like this:

```
C:\godot> dir bin/
godot.windows.editor.64.exe
```

Copy that binary to any location you like, as it contains the Project Manager, editor and all means to execute the game. However, it lacks the data to export it to the different platforms. For that the export templates are needed (which can be either downloaded from godotengine.org [https://godotengine.org/], or you can build them yourself).

Aside from that, there are a few standard options that can be set in all build targets, and which will be explained below.

Target

Target controls if the editor is contained and debug flags are used. All builds are optimized. Each mode means:

- `target=editor`: Build with editor, optimized, with debugging code (defines: `TOOLS_ENABLED`, `DEBUG_ENABLED`, `-O2`/`/O2`)
- `target=template_debug`: Build with C++ debugging symbols (defines: `DEBUG_ENABLED`, `-O2`/`/O2`)
- `target=template_release`: Build without symbols (defines: `-O3`/`/O2`)

The editor is enabled by default in all PC targets (Linux, Windows, macOS), disabled for everything else. Disabling the editor produces a binary that can run projects but does not include the editor or the Project Manager.

```
scons platform=<platform>
target=editor/template_debug/template_release
```

Development and production aliases

When creating builds for development (running debugging/profiling tools), you often have different goals compared to production builds (making binaries as fast and small as possible).

Godot provides two aliases for this purpose:

- `dev_mode=yes` is an alias for `verbose=yes warnings=extra werror=yes tests=yes`. This enables warnings-as-errors behavior (similar to Godot's continuous integration setup) and also builds unit tests so you can run them locally.
- `production=yes` is an alias for `use_static_cpp=yes debug_symbols=no lto=auto`. Statically linking libstdc++ allows for better binary portability when compiling for Linux. This alias also enables link-time optimization when compiling for Linux, Web and Windows with MinGW, but keeps LTO disabled when compiling for macOS, iOS or Windows with MSVC. This is because LTO on those platforms is very slow to link or has issues with the generated code.

You can manually override options from those aliases by specifying them on the same command line with different values. For example, you can use `scons production=yes debug_symbols=yes` to create production-optimized binaries with debugging symbols included.

Dev build

Note

`dev_build` should **not** be confused with `dev_mode`, which is an alias for several development-related options (see above).

When doing engine development the `dev_build` option can be used together with `target` to enable dev-specific code. `dev_build` defines `DEV_ENABLED`, disables optimization (`-O0`/`/0d`), enables generating debug symbols, and does not define `NDEBUG` (so `assert()` works in thirdparty libraries).

```
scons platform=<platform> dev_build=yes
```

This flag appends the `.dev` suffix (for development) to the generated binary name.

See also

There are additional SCons options to enable *sanitizers*, which are tools you can enable at compile-time to better debug certain engine issues. See Using sanitizers for more information.

Debugging symbols

By default, `debug_symbols=no` is used, which means **no** debugging symbols are included in compiled binaries. Use `debug_symbols=yes` to include debug symbols within compiled binaries, which allows debuggers and profilers to work correctly. Debugging symbols are also required for Godot's crash stacktraces to display with references to source code files and lines.

The downside is that debugging symbols are large files (significantly larger than the binaries themselves). As a result, official binaries currently do not include debugging symbols. This means you need to compile Godot yourself to have access to debugging symbols.

When using `debug_symbols=yes`, you can also use `separate_debug_symbols=yes` to put debug information in a separate file with a `.debug` suffix. This allows distributing both files independently. Note that on Windows, when compiling with MSVC, debugging information is *always* written to a separate `.pdb` file regardless of `separate_debug_symbols`.

Tip

Use the `strip <path/to/binary>` command to remove debugging symbols from a binary you've already compiled.

Optimization level

Several compiler optimization levels can be chosen from:

- `optimize=speed_trace` *(default when targeting non-Web platforms)*: Favors execution speed at the cost of larger binary size. Optimizations may sometimes negatively impact debugger usage (stack traces may be less accurate. If this occurs to you, use `optimize=debug` instead.
- `optimize=speed`: Favors even more execution speed, at the cost of even larger binary size compared to `optimize=speed_trace`. Even less friendly to debugging compared to `optimize=debug`, as this uses the most aggressive optimizations available.
- `optimize=size` *(default when targeting the Web platform)*: Favors small binaries at the cost of slower execution speed.
- `optimize=debug`: Only enables optimizations that do not impact debugging in any way. This results in faster binaries than `optimize=none`, but slower binaries than `optimize=speed_trace`.
- `optimize=none`: Do not perform any optimization. This provides the fastest build times, but the slowest execution times.
- `optimize=custom` *(advanced users only)*: Do not pass optimization arguments to the C/C++ compilers. You will have to pass arguments manually using the `cflags`, `ccflags` and `cxxflags` SCons options.

Architecture

The `arch` option is meant to control the CPU or OS version intended to run the binaries. It is focused mostly on desktop platforms and ignored everywhere else.

Supported values for the `arch` option are **auto**, **x86_32**, **x86_64**, **arm32**, **arm64**, **rv64**, **ppc32**, **ppc64** and **wasm32**.

```
scons platform=<platform> arch=
{auto|x86_32|x86_64|arm32|arm64|rv64|ppc32|ppc64|wasm32}
```

This flag appends the value of `arch` to resulting binaries when relevant. The default value `arch=auto` detects the architecture that matches the host platform.

Custom modules

It's possible to compile modules residing outside of Godot's directory tree, along with the built-in modules.

A `custom_modules` build option can be passed to the command line before compiling. The option represents a comma-separated list of directory paths containing a collection of independent C++ modules that can be seen as C++ packages, just like the built-in `modules/` directory.

For instance, it's possible to provide both relative, absolute, and user directory paths containing such modules:

```
scons
custom_modules="../modules,/abs/path/to/modules,~/src/godot_mod
ules"
```

Note

If there's any custom module with the exact directory name as a built-in module, the engine will only compile the custom one. This logic can be used to override built-in module implementations.

See also

Custom modules in C++

Cleaning generated files

Sometimes, you may encounter an error due to generated files being present. You can remove them by using `scons --clean <options>`, where `<options>` is the list of build options you've used to build Godot previously.

Alternatively, you can use `git clean -fixd` which will clean build artifacts for all platforms and configurations. Beware, as this will remove all untracked and ignored files in the repository. Don't run this command if you have uncommitted work!

Other build options

There are several other build options that you can use to configure the way Godot should be built (compiler, debug options, etc.) as well as the features to include/disable.

Check the output of `scons --help` for details about each option for the version you are willing to compile.

Overriding the build options

Using a file

The default `custom.py` file can be created at the root of the Godot Engine source to initialize any SCons build options passed via the command line:

```
# custom.py

optimize = "size"
module_mono_enabled = "yes"
use_llvm = "yes"
extra_suffix = "game_title"
```

You can also disable some of the builtin modules before compiling, saving some time it takes to build the engine. See Optimizing a build for size page for more details.

See also

You can use the online Godot build options generator [https://godot-build-options-generator.github.io/] to generate a `custom.py` file containing SCons options. You can then save this file and place it at the root of your Godot source directory.

Another custom file can be specified explicitly with the `profile` command line option, both overriding the default build configuration:

```
scons profile=path/to/custom.py
```

Note

Build options set from the file can be overridden by the command line options.

It's also possible to override the options conditionally:

```
# custom.py

import version

# Override options specific for Godot 3.x and 4.x versions.
if version.major == 3:
    pass
```

```
elif version.major == 4:
    pass
```

Using the SCONSFLAGS

`SCONSFLAGS` is an environment variable which is used by the SCons to set the options automatically without having to supply them via the command line.

For instance, you may want to force a number of CPU threads with the aforementioned `-j` option for all future builds:

Linux/macOS

```
export SCONSFLAGS="-j4"
```

Windows (cmd)

```
set SCONSFLAGS=-j4
```

Windows (PowerShell)

```
$env:SCONSFLAGS="-j4"
```

SCU (single compilation unit) build

Regular builds tend to be bottlenecked by including large numbers of headers in each compilation translation unit. Primarily to speed up development (rather than for production builds), Godot offers a "single compilation unit" build (aka "Unity / Jumbo" build).

For the folders accelerated by this option, multiple `.cpp` files are compiled in each translation unit, so headers can be shared between multiple files, which can dramatically decrease build times.

To perform an SCU build, use the `scu_build=yes` SCons option.

Note

When developing a Pull Request using SCU builds, be sure to make a regular build prior to submitting the PR. This is because SCU builds by nature include headers from earlier `.cpp` files in the translation unit, therefore won't catch all the includes you will need in a regular build. The CI will catch these errors but it will usually be faster to catch them on a local build on your machine.

Export templates

Official export templates are downloaded from the Godot Engine site: godotengine.org [https://godotengine.org/]. However, you might want to build them yourself (in case you want newer ones, you are using custom modules, or simply don't trust your own shadow).

If you download the official export templates package and unzip it, you will notice that most files are optimized binaries or packages for each platform:

```
android_debug.apk
android_release.apk
android_source.zip
ios.zip
linux_debug.arm32
linux_debug.arm64
linux_debug.x86_32
linux_debug.x86_64
linux_release.arm32
linux_release.arm64
linux_release.x86_32
linux_release.x86_64
macos.zip
version.txt
web_debug.zip
web_dlink_debug.zip
```

```
web_dlink_nothreads_debug.zip
web_dlink_nothreads_release.zip
web_dlink_release.zip
web_nothreads_debug.zip
web_nothreads_release.zip
web_release.zip
windows_debug_x86_32_console.exe
windows_debug_x86_32.exe
windows_debug_x86_64_console.exe
windows_debug_x86_64.exe
windows_debug_arm64_console.exe
windows_debug_arm64.exe
windows_release_x86_32_console.exe
windows_release_x86_32.exe
windows_release_x86_64_console.exe
windows_release_x86_64.exe
windows_release_arm64_console.exe
windows_release_arm64.exe
```

To create those yourself, follow the instructions detailed for each platform in this same tutorial section. Each platform explains how to create its own template.

The `version.txt` file should contain the corresponding Godot version identifier. This file is used to install export templates in a version-specific directory to avoid conflicts. For instance, if you are building export templates for Godot 3.1.1, `version.txt` should contain `3.1.1.stable` on the first line (and nothing else). This version identifier is based on the `major`, `minor`, `patch` (if present) and `status` lines of the version.py file in the Godot Git repository [https://github.com/godotengine/godot/blob/master/version.py].

If you are developing for multiple platforms, macOS is definitely the most convenient host platform for cross-compilation, since you can cross-compile for every target. Linux and Windows come in second place, but Linux has the advantage of being the easier platform to set this up.

Compiling for Windows

See also

This page describes how to compile Windows editor and export template binaries from source. If you're looking to export your project to Windows instead, read Exporting for Windows.

Requirements

For compiling under Windows, the following is required:

- A C++ compiler. Use one of the following:
    - Visual Studio Community [https://www.visualstudio.com/vs/community/], version 2019 or later. Visual Studio 2022 is recommended. **Make sure to enable C++ in the list of workflows to install.** If you've already installed Visual Studio without C++ support, run the installer again; it should present you a **Modify** button. Supports `x86_64`, `x86_32`, and `arm64`.
    - MinGW-w64 [https://mingw-w64.org/] with GCC can be used as an alternative to Visual Studio. Be sure to install/configure it to use the `posix` thread model. **Important:** When using MinGW to compile the `master` branch, you need GCC 9 or later. Supports `x86_64` and `x86_32` only.
    - MinGW-LLVM [https://github.com/mstorsjo/llvm-mingw/releases] with clang can be used as an alternative

to Visual Studio and MinGW-w64. Supports `x86_64`, `x86_32`, and `arm64`.

- Python 3.6+ [https://www.python.org/downloads/windows/]. **Make sure to enable the option to add Python to the `PATH` in the installer.**
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system. Using the latest release is recommended, especially for proper support of recent Visual Studio releases.

Note

If you have Scoop [https://scoop.sh/] installed, you can easily install MinGW and other dependencies using the following command:

```
scoop install gcc python scons make mingw
```

Note

If you have MSYS2 [https://www.msys2.org/] installed, you can easily install MinGW and other dependencies using the following command:

```
pacman -S mingw-w64-x86_64-python3-pip mingw-w64-x86_64-gcc  \
    mingw-w64-i686-python3-pip mingw-w64-i686-gcc make
```

For each MSYS2 MinGW subsystem, you should then run *pip3 install scons* in its shell.

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Setting up SCons

To install SCons, open the command prompt and run the following command:

```
python -m pip install scons
```

If you are prompted with the message `Defaulting to user installation because normal site-packages is not writeable`, you may have to run that command again using elevated permissions. Open a new command prompt as an Administrator then run the command again to ensure that SCons is available from the `PATH`.

To check whether you have installed Python and SCons correctly, you can type `python --version` and `scons --version` into a command prompt (`cmd.exe`).

If the commands above don't work, make sure to add Python to your `PATH` environment variable after installing it, then check again. You can do so by running the Python installer again and enabling the option to add Python to the `PATH`.

If SCons cannot detect your Visual Studio installation, it might be that your SCons version is too old. Update it to the latest version with `python -m pip install --upgrade scons`.

Downloading Godot's source

Refer to Getting the source for detailed instructions.

The tutorial will assume from now on that you placed the source code in `C:\godot`.

Warning

To prevent slowdowns caused by continuous virus scanning during compilation, add the Godot source folder to the list of exceptions in your antivirus software.

For Windows Defender, hit the `Windows` key, type "Windows Security" then hit `Enter`. Click on **Virus & threat protection** on the left panel. Under **Virus & threat protection settings** click on **Manage Settings** and scroll down to **Exclusions**. Click **Add or remove exclusions** then add the Godot source folder.

Compiling

Selecting a compiler

SCons will automatically find and use an existing Visual Studio installation. If you do not have Visual Studio installed, it will attempt to use MinGW instead. If you already have Visual Studio installed and want to use MinGW-w64, pass `use_mingw=yes` to the SCons command line. Note that MSVC builds cannot be performed from the MSYS2 or MinGW shells. Use either `cmd.exe` or PowerShell instead. If you are using MinGW-LLVM, pass both `use_mingw=yes` and `use_llvm=yes` to the SCons command line.

Tip

During development, using the Visual Studio compiler is usually a better idea, as it links the Godot binary much faster than MinGW. However, MinGW can produce more optimized binaries using link-time optimization (see below), making it a better choice for production use. This is particularly the case for the GDScript VM which performs much better with MinGW compared to MSVC. Therefore, it's recommended to use MinGW to produce builds that you distribute to players.

All official Godot binaries are built in custom containers [https://github.com/godotengine/build-containers] using MinGW.

Running SCons

After opening a command prompt, change to the root directory of the engine source code (using `cd`) and type:

```
C:\godot> scons platform=windows
```

Note

When compiling with multiple CPU threads, SCons may warn about pywin32 being missing. You can safely ignore this warning.

If all goes well, the resulting binary executable will be placed in `C:\godot\bin\` with the name `godot.windows.editor.x86_32.exe` or `godot.windows.editor.x86_64.exe`. By default, SCons will build a binary matching your CPU architecture, but this can be overridden using `arch=x86_64`, `arch=x86_32`, or `arch=arm64`.

This executable file contains the whole engine and runs without any dependencies. Running it will bring up the Project Manager.

Tip

If you are compiling Godot for production use, you can make the final executable smaller and faster by adding the SCons option `production=yes`. This enables additional compiler optimizations and link-time optimization.

LTO takes some time to run and requires up to 30 GB of available RAM while compiling (depending on toolchain). If you're running out of memory with the above option, use `production=yes lto=none` or `production=yes lto=thin` (LLVM only) for a lightweight but less effective form of LTO.

Note

If you want to use separate editor settings for your own Godot builds and official releases, you can enable Self-contained mode by creating a file called `._sc_` or `_sc_` in the `bin/` folder.

Compiling with support for Direct3D 12

By default, builds of Godot do not contain support for the Direct3D 12 graphics API.

To compile Godot with Direct3D 12 support you need at least the following item:

- godot-nir-static library [https://github.com/godotengine/godot-nir-static/releases/]. We compile the Mesa libraries you will need into a static library. Download it anywhere, unzip it and remember the path to the unzipped folder, you will need it below.

Note

You can optionally build the godot-nir-static libraries yourself with the following steps:

1. Install the Python package mako [https://www.makotemplates.org] which is needed to generate some files.
2. Clone the godot-nir-static [https://github.com/godotengine/godot-nir-static] directory and navigate to it.
3. Run the following:

   ```
   git submodule update --init
   ./update_mesa.sh
   scons
   ```

   If you are building with MinGW-w64, add `use_mingw=yes` to the `scons` command, you can also specify build architecture using `arch={architecture}`. If you are building with MinGW-LLVM, add both `use_mingw=yes` and `use_llvm=yes` to the `scons` command.

   If you are building with MinGW and the binaries are not located in the `PATH`, add `mingw_prefix="/path/to/mingw"` to the `scons` command.

   Mesa static library should be built using the same compiler and the same CRT (if you are building with MinGW) you are using for building Godot.

Optionally, you can compile with the following for additional features:

- PIX [https://devblogs.microsoft.com/pix/download] is a performance tuning and debugging application for Direct3D12 applications. If you compile-in support for it, you can get much more detailed information through PIX that will help you optimize your game and troubleshoot graphics bugs. To use it, download the WinPixEventRuntime package. You will be taken to a NuGet package page where you can click "Download package" to get it. Once downloaded, change the file extension to .zip and unzip the file to some path.

- Agility SDK [https://devblogs.microsoft.com/directx/directx12agility] can be used to provide access to the latest Direct3D 12 features without relying on driver updates. To use it, download the latest Agility SDK package. You will be taken to a NuGet package page where you can click "Download package" to get it. Once downloaded, change the file extension to .zip and unzip the file to some path.

Note

If you use a preview version of the Agility SDK, remember to enable developer mode in Windows; otherwise it won't be used.

Note

If you want to use a PIX with MinGW build, navigate to PIX runtime directory and use the following commands to generate import library:

```
# For x86-64:
gendef ./bin/x64/WinPixEventRuntime.dll
dlltool --machine i386:x86-64 --no-leading-underscore -d
WinPixEventRuntime.def -D WinPixEventRuntime.dll -l
./bin/x64/libWinPixEventRuntime.a

# For ARM64:
gendef ./bin/ARM64/WinPixEventRuntime.dll
dlltool --machine arm64 --no-leading-underscore -d
WinPixEventRuntime.def -D WinPixEventRuntime.dll -l
./bin/ARM64/libWinPixEventRuntime.a
```

When building Godot, you will need to tell SCons to use Direct3D 12 and where to look for the additional libraries:

```
C:\godot> scons platform=windows d3d12=yes mesa_libs=<...>
```

Or, with all options enabled:

```
C:\godot> scons platform=windows d3d12=yes mesa_libs=<...>
agility_sdk_path=<...> pix_path=<...>
```

Note

For the Agility SDK's DLLs you have to explicitly choose the kind of workflow. Single-arch is the default (DLLs copied to `bin/`). If you pass `agility_sdk_multi_arch=yes` to SCons, you'll opt-in for multi-arch. DLLs will be copied to the appropriate `bin/<arch>/` subdirectories and at runtime the right one will be loaded.

Compiling with ANGLE support

ANGLE provides a translation layer from OpenGL ES 3.x to Direct3D 11 and can be used to improve support for the Compatibility renderer on some older GPUs with outdated OpenGL drivers and on Windows for ARM.

By default, Godot is built with dynamically linked ANGLE, you can use it by placing `libEGL.dll` and `libGLESv2.dll` alongside the executable.

Note

You can use dynamically linked ANGLE with export templates as well, rename aforementioned DLLs to `libEGL.{architecture}.dll` and `libGLESv2.{architecture}.dll` and place them alongside export template executables, and libraries will be automatically copied during the export process.

To compile Godot with statically linked ANGLE:

- Download pre-built static libraries from godot-angle-static library [https://github.com/godotengine/godot-angle-static/releases], and unzip them.
- When building Godot, add `angle_libs={path}` to tell SCons where to look for the ANGLE libraries:

  ```
  scons platform=windows angle_libs=<...>
  ```

Note

You can optionally build the godot-angle-static libraries yourself with the following steps:

1. Clone the godot-angle-static [https://github.com/godotengine/godot-angle-static] directory and navigate to it.
2. Run the following command:

   ```
   git submodule update --init
   ./update_angle.sh
   scons
   ```

   If you are buildng with MinGW, add `use_mingw=yes` to the command, you can also specify build architecture using `arch={architecture}`. If you are building with MinGW-LLVM, add both `use_mingw=yes` and `use_llvm=yes` to the `scons` command.

   If you are building with MinGW and the binaries are not located in the `PATH`, add `mingw_prefix="/path/to/mingw"` to the `scons` command.

ANGLE static library should be built using the same compiler and the same CRT (if you are building with MinGW) you are using for building Godot.

Development in Visual Studio

Using an IDE is not required to compile Godot, as SCons takes care of everything. But if you intend to do engine development or debugging of the engine's C++ code, you may be interested in configuring a code editor or an IDE.

Folder-based editors don't require any particular setup to start working with Godot's codebase. To edit projects with Visual Studio they need to be set up as a solution.

You can create a Visual Studio solution via SCons by running SCons with the `vsproj=yes` parameter, like this:

```
scons platform=windows vsproj=yes
```

You will be able to open Godot's source in a Visual Studio solution now, and able to build Godot using Visual Studio's **Build** button.

See also

See Visual Studio for further details.

Cross-compiling for Windows from other operating systems

If you are a Linux or macOS user, you need to install MinGW-w64 [https://www.mingw-w64.org/], which typically comes in 32-bit and 64-bit variants, or MinGW-LLVM [https://github.com/mstorsjo/llvm-mingw/releases], which comes as a single archive for all target architectures. The package names may differ based on your distribution, here are some known ones:

Arch Linux

```
pacman -Sy mingw-w64
```

Debian / Ubuntu

```
apt install mingw-w64
```

Fedora

```
dnf install mingw64-gcc-c++ mingw64-winpthreads-static \
            mingw32-gcc-c++ mingw32-winpthreads-static
```

macOS

```
brew install mingw-w64
```

Mageia

```
urpmi mingw64-gcc-c++ mingw64-winpthreads-static \
      mingw32-gcc-c++ mingw32-winpthreads-static
```

Before attempting the compilation, SCons will check for the following binaries in your `PATH` environment variable:

```
# for MinGW-w64
i686-w64-mingw32-gcc
x86_64-w64-mingw32-gcc

# for MinGW-LLVM
aarch64-w64-mingw32-clang
```

```
i686-w64-mingw32-clang
x86_64-w64-mingw32-clang
```

If the binaries are not located in the `PATH` (e.g. `/usr/bin`), you can define the following environment variable to give a hint to the build system:

```
export MINGW_PREFIX="/path/to/mingw"
```

Where `/path/to/mingw` is the path containing the `bin` directory where `i686-w64-mingw32-gcc` and `x86_64-w64-mingw32-gcc` are located (e.g. `/opt/mingw-w64` if the binaries are located in `/opt/mingw-w64/bin`).

To make sure you are doing things correctly, executing the following in the shell should result in a working compiler (the version output may differ based on your system):

```
${MINGW_PREFIX}/bin/x86_64-w64-mingw32-gcc --version
# x86_64-w64-mingw32-gcc (GCC) 13.2.0
```

Note

If you are building with MinGW-LLVM, add `use_llvm=yes` to the `scons` command.

Note

When cross-compiling for Windows using MinGW-w64, keep in mind only `x86_64` and `x86_32` architectures are supported. MinGW-LLVM supports `arm64` as well. Be sure to specify the right `arch=` option when invoking SCons if building from a different architecture.

Troubleshooting

Cross-compiling from some Ubuntu versions may lead to this bug [https://github.com/godotengine/godot/issues/9258], due to a default configuration lacking support for POSIX threading.

You can change that configuration following those instructions, for 64-bit:

```
sudo update-alternatives --config x86_64-w64-mingw32-gcc
<choose x86_64-w64-mingw32-gcc-posix from the list>
sudo update-alternatives --config x86_64-w64-mingw32-g++
<choose x86_64-w64-mingw32-g++-posix from the list>
```

And for 32-bit:

```
sudo update-alternatives --config i686-w64-mingw32-gcc
<choose i686-w64-mingw32-gcc-posix from the list>
sudo update-alternatives --config i686-w64-mingw32-g++
<choose i686-w64-mingw32-g++-posix from the list>
```

Creating Windows export templates

Windows export templates are created by compiling Godot without the editor, with the following flags:

```
C:\godot> scons platform=windows target=template_debug
arch=x86_32
C:\godot> scons platform=windows target=template_release
arch=x86_32
C:\godot> scons platform=windows target=template_debug
arch=x86_64
C:\godot> scons platform=windows target=template_release
arch=x86_64
C:\godot> scons platform=windows target=template_debug
arch=arm64
C:\godot> scons platform=windows target=template_release
arch=arm64
```

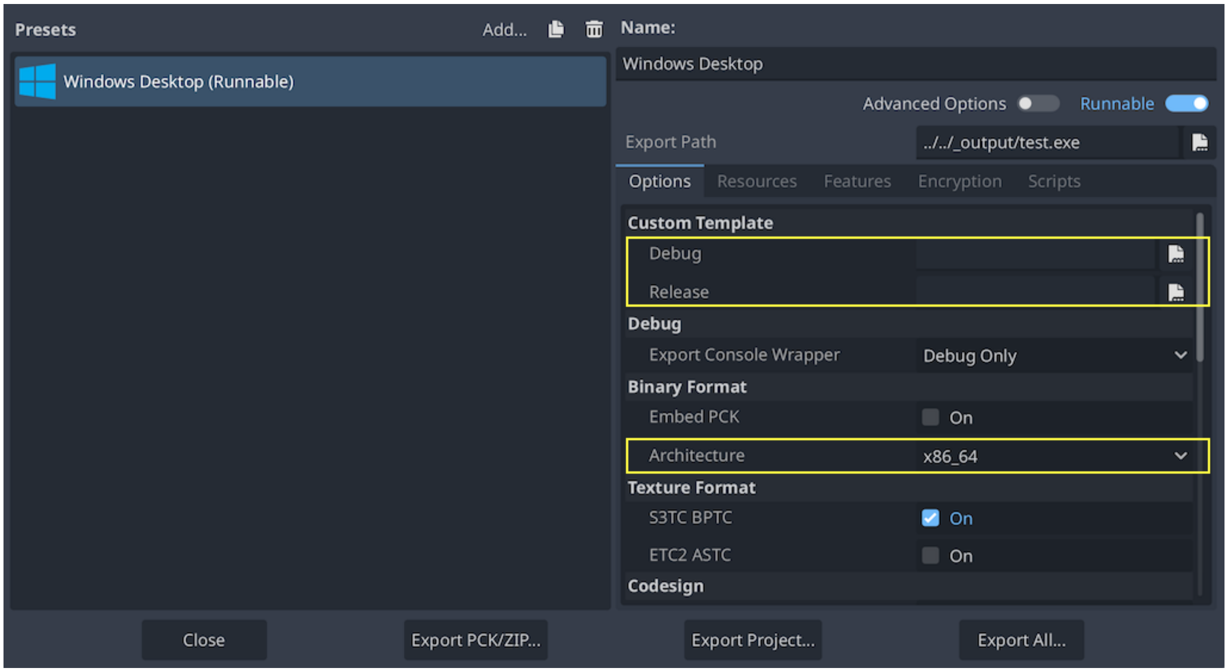
If you plan on replacing the standard export templates, copy these to the following location, replacing `<version>` with the version identifier (such as `4.2.1.stable` or `4.3.dev`):

```
%APPDATA%\Godot\export_templates\<version>\
```

With the following names:

```
windows_debug_x86_32_console.exe
windows_debug_x86_32.exe
windows_debug_x86_64_console.exe
windows_debug_x86_64.exe
windows_debug_arm64_console.exe
windows_debug_arm64.exe
windows_release_x86_32_console.exe
windows_release_x86_32.exe
windows_release_x86_64_console.exe
windows_release_x86_64.exe
windows_release_arm64_console.exe
windows_release_arm64.exe
```

However, if you are using custom modules or custom engine code, you may instead want to configure your binaries as custom export templates here:



Presets

Windows Desktop (Runnable)

Name: Windows Desktop

Advanced Options ○ Runnable ●

Export Path: ../../_output/test.exe

Custom Template

Debug:

Release:

Debug

Export Console Wrapper: Debug Only

Binary Format

Embed PCK: ☐ On

Architecture: x86_64

Texture Format

S3TC BPTC: ☑ On

ETC2 ASTC: ☐ On

Codesign

Select matching architecture in the export config.

You don't need to copy them in this case, just reference the resulting files in the `bin\` directory of your Godot source folder, so the next time you build, you will automatically have the custom templates referenced.

Compiling for Linux, *BSD

See also

This page describes how to compile Linux editor and export template binaries from source. If you're looking to export your project to Linux instead, read Exporting for Linux.

Requirements

For compiling under Linux or other Unix variants, the following is required:

- GCC 9+ or Clang 6+.
- Python 3.6+ [https://www.python.org/downloads/].
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system.
- pkg-config (used to detect the development libraries listed below).
- Development libraries:
  - X11, Xcursor, Xinerama, Xi and XRandR.
  - Wayland and wayland-scanner.
  - Mesa.
  - ALSA.
  - PulseAudio.
- *Optional* - libudev (build with `udev=yes`).

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Distro-specific one-liners

Alpine Linux

```
apk add \
  scons \
  pkgconf \
  gcc \
  g++ \
  libx11-dev \
  libxcursor-dev \
  libxinerama-dev \
  libxi-dev \
  libxrandr-dev \
  mesa-dev \
  eudev-dev \
  alsa-lib-dev \
  pulseaudio-dev
```

Arch Linux

```
pacman -Sy --noconfirm --needed \
  scons \
  pkgconf \
  gcc \
  libxcursor \
  libxinerama \
  libxi \
  libxrandr \
  wayland-utils \
  mesa \
  glu \
  libglvnd \
  alsa-lib \
  pulseaudio
```

Debian/Ubuntu

```
sudo apt-get update
sudo apt-get install -y \
  build-essential \
  scons \
  pkg-config \
  libx11-dev \
  libxcursor-dev \
  libxinerama-dev \
  libgl1-mesa-dev \
  libglu1-mesa-dev \
  libasound2-dev \
  libpulse-dev \
  libudev-dev \
  libxi-dev \
  libxrandr-dev \
  libwayland-dev
```

Fedora

```
sudo dnf install -y \
  scons \
  pkgconfig \
  libX11-devel \
  libXcursor-devel \
  libXrandr-devel \
  libXinerama-devel \
  libXi-devel \
  wayland-devel \
  mesa-libGL-devel \
  mesa-libGLU-devel \
  alsa-lib-devel \
  pulseaudio-libs-devel \
  libudev-devel \
  gcc-c++ \
  libstdc++-static \
  libatomic-static
```

FreeBSD

```
pkg install \
  py37-scons \
  pkgconf \
  xorg-libraries \
  libXcursor \
  libXrandr \
  libXi \
  xorgproto \
  libGLU \
  alsa-lib \
  pulseaudio
```

Gentoo

```
emerge --sync
emerge -an \
  dev-build/scons \
  x11-libs/libX11 \
  x11-libs/libXcursor \
  x11-libs/libXinerama \
  x11-libs/libXi \
  dev-util/wayland-scanner \
  media-libs/mesa \
  media-libs/glu \
  media-libs/alsa-lib \
  media-sound/pulseaudio
```

Mageia

```
sudo urpmi --auto \
  scons \
  task-c++-devel \
  wayland-devel \
  "pkgconfig(alsa)" \
  "pkgconfig(glu)" \
  "pkgconfig(libpulse)" \
  "pkgconfig(udev)" \
  "pkgconfig(x11)" \
  "pkgconfig(xcursor)" \
  "pkgconfig(xinerama)" \
  "pkgconfig(xi)" \
  "pkgconfig(xrandr)"
```

NetBSD

```
pkg_add \
  pkg-config \
  py37-scons
```

Hint

For audio support, you can optionally install `pulseaudio`.

OpenBSD

```
pkg_add \
  python \
  scons \
  llvm
```

openSUSE

```
sudo zypper install -y \
  scons \
  pkgconfig \
  libX11-devel \
  libXcursor-devel \
  libXrandr-devel \
  libXinerama-devel \
  libXi-devel \
  wayland-devel \
  Mesa-libGL-devel \
  alsa-devel \
  libpulse-devel \
  libudev-devel \
  gcc-c++ \
  libGLU1
```

Solus

```
eopkg install -y \
  -c system.devel \
  scons \
  libxcursor-devel \
  libxinerama-devel \
  libxi-devel \
  libxrandr-devel \
  wayland-devel \
  mesalib-devel \
  libglu \
  alsa-lib-devel \
  pulseaudio-devel
```

Compiling

Start a terminal, go to the root dir of the engine source code and type:

```
scons platform=linuxbsd
```

Note

Prior to Godot 4.0, the Linux/*BSD target was called `x11` instead of `linuxbsd`. If you are looking to compile Godot 3.x, make sure to use the 3.x branch of this documentation [https://docs.godotengine.org/en/3.6/development/compiling/compiling_for_x11.html].

If all goes well, the resulting binary executable will be placed in the "bin" subdirectory. This executable file contains the whole engine and runs without any dependencies. Executing it will bring up the Project Manager.

Note

If you wish to compile using Clang rather than GCC, use this command:

```
scons platform=linuxbsd use_llvm=yes
```

Using Clang appears to be a requirement for OpenBSD, otherwise fonts would not build. For RISC-V architecture devices, use the Clang compiler instead of the GCC compiler.

Tip

If you are compiling Godot for production use, you can make the final executable smaller and faster by adding the SCons option `production=yes`. This enables additional compiler optimizations and link-time optimization.

LTO takes some time to run and requires about 7 GB of available RAM while compiling. If you're running out of memory with the above option, use `production=yes lto=none` or `production=yes lto=thin` for a lightweight but less effective form of LTO.

Note

If you want to use separate editor settings for your own Godot builds and official releases, you can enable Self-contained mode by creating a file called `._sc_` or `_sc_` in the `bin/` folder.

Running a headless/server build

To run in *headless* mode which provides editor functionality to export projects in an automated manner, use the normal build:

```
scons platform=linuxbsd target=editor
```

And then use the `--headless` command line argument:

```
./bin/godot.linuxbsd.editor.x86_64 --headless
```

To compile a debug *server* build which can be used with remote debugging tools, use:

```
scons platform=linuxbsd target=template_debug
```

To compile a *server* build which is optimized to run dedicated game servers, use:

```
scons platform=linuxbsd target=template_release production=yes
```

Building export templates

Warning

Linux binaries usually won't run on distributions that are older than the distribution they were built on. If you wish to distribute binaries that work on most distributions, you should build them on an old distribution such as Ubuntu 16.04. You can use a virtual machine or a container to set up a suitable build environment.

To build Linux or *BSD export templates, run the build system with the following parameters:

- (32 bits)

```
scons platform=linuxbsd target=template_release arch=x86_32
scons platform=linuxbsd target=template_debug arch=x86_32
```

- (64 bits)

```
scons platform=linuxbsd target=template_release arch=x86_64
scons platform=linuxbsd target=template_debug arch=x86_64
```

Note that cross-compiling for the opposite bits (64/32) as your host platform is not always straight-forward and might need a chroot environment.

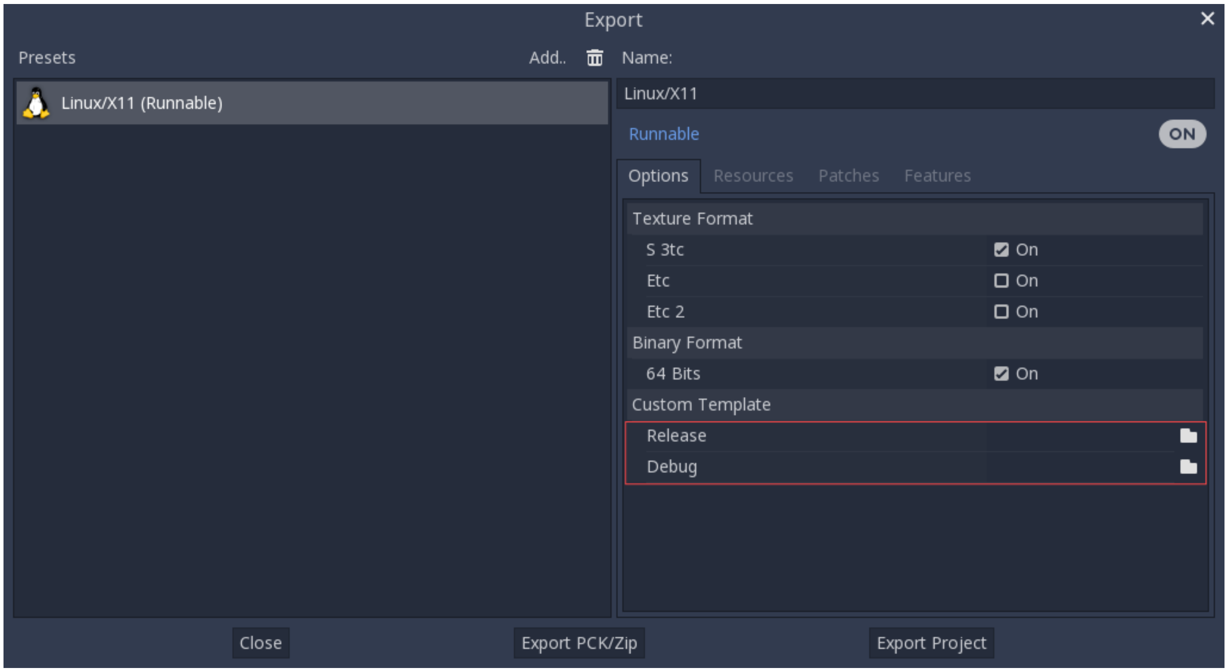
To create standard export templates, the resulting files in the `bin/` folder must be copied to:

```
$HOME/.local/share/godot/export_templates/<version>/
```

and named like this (even for *BSD which is seen as "Linux/X11" by Godot):

```
linux_debug.arm32
linux_debug.arm64
linux_debug.x86_32
linux_debug.x86_64
linux_release.arm32
linux_release.arm64
linux_release.x86_32
linux_release.x86_64
```

However, if you are writing your custom modules or custom C++ code, you might instead want to configure your binaries as custom export templates here:


Export
Presets
Add..
Linux/X11 (Runnable)
Options
Resources
Patches
Features
Close
Export PCK/Zip
Export Project

Name: Linux/X11

Runnable ON

Texture Format
S 3tc ☑ On
Etc ☐ On
Etc 2 ☐ On
Binary Format
64 Bits ☑ On
Custom Template
Release
Debug

You don't even need to copy them, you can just reference the resulting files in the `bin/` directory of your Godot source folder, so the next time you build, you automatically have the custom templates referenced.

Cross-compiling for RISC-V devices

To cross-compile Godot for RISC-V devices, we need to setup the following items:

- riscv-gnu-toolchain [https://github.com/riscv-collab/riscv-gnu-toolchain/releases]. While we are not going to use this directly, it provides us with a sysroot, as well as header and libraries files that we will need. There are many versions to choose from, however, the older the toolchain, the more compatible our final binaries will be. If in doubt, use this version [https://github.com/riscv-collab/riscv-gnu-toolchain/releases/tag/2021.12.22], and download `riscv64-glibc-ubuntu-18.04-nightly-2021.12.22-nightly.tar.gz`. Extract it somewhere and remember its path.

- Clang. RISC-V GCC has bugs with its atomic operations [https://github.com/riscv-collab/riscv-gcc/issues/15] which prevent it from compiling Godot correctly. Any version of Clang from 16.0.0 upwards will suffice. Download it from the package manager of your distro, and make sure that it *can* compile to RISC-V. You can verify by executing this command `clang -print-targets`, make sure you see `riscv64` on the list of targets.
- mold [https://github.com/rui314/mold/releases]. This fast linker, is the only one that correctly links the resulting binary. Download it, extract it, and make sure to add its `bin` folder to your PATH. Run `mold --help | grep support` to check if your version of Mold supports RISC-V. If you don't see RISC-V, your Mold may need to be updated.

To make referencing our toolchain easier, we can set an environment variable like this:

```
export RISCV_TOOLCHAIN_PATH="path to toolchain here"
```

This way, we won't have to manually set the directory location each time we want to reference it.

With all the above setup, we are now ready to build Godot.

Go to the root of the source code, and execute the following build command:

```
scons arch=rv64 use_llvm=yes linker=mold lto=none target=editor \
    ccflags="--sysroot=$RISCV_TOOLCHAIN_PATH/sysroot --gcc-toolchain=$RISCV_TOOLCHAIN_PATH -target riscv64-unknown-linux-gnu" \
    linkflags="--sysroot=$RISCV_TOOLCHAIN_PATH/sysroot --gcc-toolchain=$RISCV_TOOLCHAIN_PATH -target riscv64-unknown-linux-gnu"
```

The command is similar in nature, but with some key changes. `ccflags` and `linkflags` append additional flags to the build. `--`

`sysroot` points to a folder simulating a Linux system, it contains all the headers, libraries, and `.so` files Clang will use. `--gcc-toolchain` tells Clang where the complete toolchain is, and `-target riscv64-unknown-linux-gnu` indicates to Clang the target architecture, and OS we want to build for.

If all went well, you should now see a `bin` directory, and within it, a binary similar to the following:

```
godot.linuxbsd.editor.rv64.llvm
```

You can now copy this executable to your favorite RISC-V device, then launch it there by double-clicking, which should bring up the project manager.

If you later decide to compile the export templates, copy the above build command but change the value of `target` to `template_debug` for a debug build, or `template_release` for a release build.

Using Clang and LLD for faster development

You can also use Clang and LLD to build Godot. This has two upsides compared to the default GCC + GNU ld setup:

- LLD links Godot significantly faster compared to GNU ld or gold. This leads to faster iteration times.
- Clang tends to give more useful error messages compared to GCC.

To do so, install Clang and the `lld` package from your distribution's package manager then use the following SCons command:

```
scons platform=linuxbsd use_llvm=yes linker=lld
```

After the build is completed, a new binary with a `.llvm` suffix will be created in the `bin/` folder.

It's still recommended to use GCC for production builds as they can be compiled using link-time optimization, making the resulting binaries smaller and faster.

If this error occurs:

```
/usr/bin/ld: cannot find -l:libatomic.a: No such file or
directory
```

There are two solutions:

- In your SCons command, add the parameter `use_static_cpp=no`.
- Follow these instructions [https://github.com/ivmai/libatomic_ops#installation-and-usage] to configure, build, and install `libatomic_ops`. Then, copy `/usr/lib/libatomic_ops.a` to `/usr/lib/libatomic.a`, or create a soft link to `libatomic_ops` by command `ln -s /usr/lib/libatomic_ops.a /usr/lib/libatomic.a`. The soft link can ensure the latest `libatomic_ops` will be used without the need to copy it every time when it is updated.

Using mold for faster development

For even faster linking compared to LLD, you can use mold [https://github.com/rui314/mold]. mold can be used with either GCC or Clang.

As of January 2023, mold is not readily available in Linux distribution repositories, so you will have to install its binaries manually.

- Download mold binaries from its releases page [https://github.com/rui314/mold/releases/latest].

- Extract the `.tar.gz` file, then move the extracted folder to a location such as `.local/share/mold`.
- Add `$HOME/.local/share/mold/bin` to your user's `PATH` environment variable. For example, you can add the following line at the end of your `$HOME/.bash_profile` file:

```
PATH="$HOME/.local/share/mold/bin:$PATH"
```

- Open a new terminal (or run `source "$HOME/.bash_profile"`), then use the following SCons command when compiling Godot:

  ```
  scons platform=linuxbsd linker=mold
  ```

Using system libraries for faster development

Godot bundles the source code of various third-party libraries. [https://github.com/godotengine/godot/tree/master/thirdparty] You can choose to use system versions of third-party libraries instead. This makes the Godot binary faster to link, as third-party libraries are dynamically linked. Therefore, they don't need to be statically linked every time you build the engine (even on small incremental changes).

However, not all Linux distributions have packages for third-party libraries available (or they may not be up-to-date).

Moving to system libraries can reduce linking times by several seconds on slow CPUs, but it requires manual testing depending on your Linux distribution. Also, you may not be able to use system libraries for everything due to bugs in the system library packages (or in the build system, as this feature is less tested).

To compile Godot with system libraries, install these dependencies **on top** of the ones listed in the Distro-specific one-liners:

Debian/Ubuntu

```
sudo apt-get update
sudo apt-get install -y \
  libembree-dev \
  libenet-dev \
  libfreetype-dev \
  libpng-dev \
  zlib1g-dev \
  libgraphite2-dev \
  libharfbuzz-dev \
  libogg-dev \
  libtheora-dev \
  libvorbis-dev \
  libwebp-dev \
  libmbedtls-dev \
  libminiupnpc-dev \
  libpcre2-dev \
  libzstd-dev \
  libsquish-dev \
  libicu-dev
```

Fedora

```
sudo dnf install -y \
  embree3-devel \
  enet-devel \
  glslang-devel \
  graphite2-devel \
  harfbuzz-devel \
  libicu-devel \
  libsquish-devel \
  libtheora-devel \
  libvorbis-devel \
  libwebp-devel \
  libzstd-devel \
  mbedtls-devel \
  miniupnpc-devel
```

After installing all required packages, use the following command to build Godot:

```
scons platform=linuxbsd builtin_embree=no builtin_enet=no
builtin_freetype=no builtin_graphite=no builtin_harfbuzz=no
builtin_libogg=no builtin_libpng=no builtin_libtheora=no
builtin_libvorbis=no builtin_libwebp=no builtin_mbedtls=no
builtin_miniupnpc=no builtin_pcre2=no builtin_zlib=no
builtin_zstd=no
```

On Debian stable, you will need to remove *builtin_embree=no* as the system-provided Embree version is too old to work with Godot's latest *master* branch (which requires Embree 4).

You can view a list of all built-in libraries that have system alternatives by running `scons -h`, then looking for options starting with `builtin_`.

Warning

When using system libraries, the resulting library is **not** portable across Linux distributions anymore. Do not use this approach for creating binaries you intend to distribute to others, unless you're creating a package for a Linux distribution.

Using Pyston for faster development

You can use Pyston [https://www.pyston.org/] to run SCons. Pyston is a JIT-enabled implementation of the Python language (which SCons is written in). It is currently only compatible with Linux. Pyston can speed up incremental builds significantly, often by a factor between 1.5× and 2×. Pyston can be combined with Clang and LLD to get even faster builds.

- Download the latest portable Pyston release [https://github.com/pyston/pyston/releases/latest].

- Extract the portable `.tar.gz` to a set location, such as `$HOME/.local/opt/pyston/` (create folders as needed).
- Use `cd` to reach the extracted Pyston folder from a terminal, then run `./pyston -m pip install scons` to install SCons within Pyston.
- To make SCons via Pyston easier to run, create a symbolic link of its wrapper script to a location in your `PATH` environment variable:

  ```
  ln -s ~/.local/opt/pyston/bin/scons ~/.local/bin/pyston-scons
  ```

- Instead of running `scons <build arguments>`, run `pyston-scons <build arguments>` to compile Godot.

If you can't run `pyston-scons` after creating the symbolic link, make sure `$HOME/.local/bin/` is part of your user's `PATH` environment variable.

Note

Alternatively, you can run `python -m pip install pyston_lite_autoload` then run SCons as usual. This will automatically load a subset of Pyston's optimizations in any Python program you run. However, this won't bring as much of a performance improvement compared to installing "full" Pyston.

Compiling for macOS

Note

This page describes how to compile macOS editor and export template binaries from source. If you're looking to export your project to macOS instead, read Exporting for macOS.

Requirements

For compiling under macOS, the following is required:

- Python 3.6+ [https://www.python.org/downloads/macos/].
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system.
- Xcode [https://apps.apple.com/us/app/xcode/id497799835] (or the more lightweight Command Line Tools for Xcode).
- Vulkan SDK [https://sdk.lunarg.com/sdk/download/latest/mac/vulkan-sdk.dmg] for MoltenVK (macOS doesn't support Vulkan out of the box).

Note

If you have Homebrew [https://brew.sh/] installed, you can easily install SCons using the following command:

```
brew install scons
```

Installing Homebrew will also fetch the Command Line Tools for Xcode automatically if you don't have them already.

Similarly, if you have MacPorts [https://www.macports.org/] installed, you can easily install SCons using the following command:

```
sudo port install scons
```

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Compiling

Start a terminal, go to the root directory of the engine source code.

To compile for Intel (x86-64) powered Macs, use:

```
scons platform=macos arch=x86_64
```

To compile for Apple Silicon (ARM64) powered Macs, use:

```
scons platform=macos arch=arm64
```

To support both architectures in a single "Universal 2" binary, run the above two commands and then use `lipo` to bundle them together:

```
lipo -create bin/godot.macos.editor.x86_64
bin/godot.macos.editor.arm64 -output
bin/godot.macos.editor.universal
```

If all goes well, the resulting binary executable will be placed in the `bin/` subdirectory. This executable file contains the whole engine and runs without any dependencies. Executing it will bring up the Project Manager.

Note

If you want to use separate editor settings for your own Godot builds and official releases, you can enable Self-contained mode by creating a file called `._sc_` or `_sc_` in the `bin/` folder.

To create an `.app` bundle like in the official builds, you need to use the template located in `misc/dist/macos_tools.app`. Typically, for an optimized editor binary built with `dev_build=yes`:

```
cp -r misc/dist/macos_tools.app ./Godot.app
mkdir -p Godot.app/Contents/MacOS
cp bin/godot.macos.editor.universal
Godot.app/Contents/MacOS/Godot
chmod +x Godot.app/Contents/MacOS/Godot
codesign --force --timestamp --options=runtime --entitlements
misc/dist/macos/editor.entitlements -s - Godot.app
```

Note

If you are building the `master` branch, you also need to include support for the MoltenVK Vulkan portability library. By default, it will be linked statically from your installation of the Vulkan SDK for macOS. You can also choose to link it dynamically by passing `use_volk=yes` and including the dynamic library in your `.app` bundle:

```
mkdir -p Godot.app/Contents/Frameworks
cp <Vulkan SDK path>/macOS/lib/libMoltenVK.dylib
Godot.app/Contents/Frameworks/libMoltenVK.dylib
```

Running a headless/server build

To run in *headless* mode which provides editor functionality to export projects in an automated manner, use the normal build:

```
scons platform=macos target=editor
```

And then use the `--headless` command line argument:

```
./bin/godot.macos.editor.x86_64 --headless
```

To compile a debug *server* build which can be used with remote debugging tools, use:

```
scons platform=macos target=template_debug
```

To compile a release *server* build which is optimized to run dedicated game servers, use:

```
scons platform=macos target=template_release production=yes
```

Building export templates

To build macOS export templates, you have to compile using the targets without the editor: `target=template_release` (release template) and `target=template_debug`.

Official templates are universal binaries which support both Intel x86_64 and ARM64 architectures. You can also create export templates that support only one of those two architectures by leaving out the `lipo` step below.

- For Intel x86_64:

  ```
  scons platform=macos target=template_release arch=x86_64
  scons platform=macos target=template_debug arch=x86_64
  ```

- For Arm64 (Apple M1):

  ```
  scons platform=macos target=template_release arch=arm64
  scons platform=macos target=template_debug arch=arm64
  ```

To support both architectures in a single "Universal 2" binary, run the above two commands blocks and then use `lipo` to bundle them together:

```
lipo -create bin/godot.macos.template_release.x86_64
bin/godot.macos.template_release.arm64 -output
bin/godot.macos.template_release.universal
lipo -create bin/godot.macos.template_debug.x86_64
bin/godot.macos.template_debug.arm64 -output
bin/godot.macos.template_debug.universal
```

To create an `.app` bundle like in the official builds, you need to use the template located in `misc/dist/macos_template.app`. The release and debug builds should be placed in `macos_template.app/Contents/MacOS` with the names `godot_macos_release.universal` and `godot_macos_debug.universal` respectively. You can do so with the following commands (assuming a universal build, otherwise replace the `.universal` extension with the one of your arch-specific binaries):

```
cp -r misc/dist/macos_template.app .
mkdir -p macos_template.app/Contents/MacOS
cp bin/godot.macos.template_release.universal
macos_template.app/Contents/MacOS/godot_macos_release.universal
cp bin/godot.macos.template_debug.universal
macos_template.app/Contents/MacOS/godot_macos_debug.universal
chmod +x macos_template.app/Contents/MacOS/godot_macos*
```

Note

If you are building the `master` branch, you also need to include support for the MoltenVK Vulkan portability library. By default, it will be linked statically from your installation of the Vulkan SDK for macOS. You can also choose to link it dynamically by passing `use_volk=yes` and including the dynamic library in your `.app` bundle:

```
mkdir -p macos_template.app/Contents/Frameworks
cp <Vulkan SDK path>/macOS/libs/libMoltenVK.dylib
```

```
macos_template.app/Contents/Frameworks/libMoltenVK.dylib
```

You can then zip the `macos_template.app` folder to reproduce the `macos.zip` template from the official Godot distribution:

```
zip -q -9 -r macos.zip macos_template.app
```

Using Pyston for faster development

You can use Pyston [https://www.pyston.org/] to run SCons. Pyston is a JIT-enabled implementation of the Python language (which SCons is written in). Its "full" version is currently only compatible with Linux, but Pyston-lite is also compatible with macOS (both x86 and ARM). Pyston can speed up incremental builds significantly, often by a factor between 1.5× and 2×. Pyston can be combined with alternative linkers such as LLD or Mold to get even faster builds.

To install Pyston-lite, run `python -m pip install pyston_lite_autoload` then run SCons as usual. This will automatically load a subset of Pyston's optimizations in any Python program you run. However, this won't bring as much of a performance improvement compared to installing "full" Pyston (which currently can't be done on macOS).

Cross-compiling for macOS from Linux

It is possible to compile for macOS in a Linux environment (and maybe also in Windows using the Windows Subsystem for Linux). For that, you'll need to install OSXCross [https://github.com/tpoechtrager/osxcross] to be able to use macOS as a target. First, follow the instructions to install it:

Clone the OSXCross repository [https://github.com/tpoechtrager/osxcross] somewhere on your machine (or download a ZIP file and extract it somewhere), e.g.:

```
git clone --depth=1
https://github.com/tpoechtrager/osxcross.git "$HOME/osxcross"
```

1. Follow the instructions to package the SDK: https://github.com/tpoechtrager/osxcross#packaging-the-sdk
2. Follow the instructions to install OSXCross: https://github.com/tpoechtrager/osxcross#installation

After that, you will need to define the `OSXCROSS_ROOT` as the path to the OSXCross installation (the same place where you cloned the repository/extracted the zip), e.g.:

```
export OSXCROSS_ROOT="$HOME/osxcross"
```

Now you can compile with SCons like you normally would:

```
scons platform=macos
```

If you have an OSXCross SDK version different from the one expected by the SCons buildsystem, you can specify a custom one with the `osxcross_sdk` argument:

```
scons platform=macos osxcross_sdk=darwin15
```

Compiling for Android

See also

This page describes how to compile Android export template binaries from source. If you're looking to export your project to Android instead, read Exporting for Android.

Note

In most cases, using the built-in deployer and export templates is good enough. Compiling the Android APK manually is mostly useful for custom builds or custom packages for the deployer.

Also, you still need to follow the steps mentioned in the Exporting for Android tutorial before attempting to build a custom export template.

Requirements

For compiling under Windows, Linux or macOS, the following is required:

- Python 3.6+ [https://www.python.org/downloads/].
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system.
- Android SDK [https://developer.android.com/studio/#command-tools] (command-line tools are sufficient).

  - Required SDK components will be automatically installed.
  - On Linux, **do not use an Android SDK provided by your distribution's repositories** as it will often be outdated.
  - On macOS, **do not use an Android SDK provided by Homebrew** as it will not be installed in a unified location.

- Gradle (will be downloaded and installed automatically if missing).

- JDK 17 (either OpenJDK or Oracle JDK).

  - You can download a build from Adoptium [https://adoptium.net/temurin/releases/?variant=openjdk17].

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Setting up the buildsystem

- Set the environment variable `ANDROID_HOME` to point to the Android SDK. If you downloaded the Android command-line tools, this would be the folder where you extracted the contents of the ZIP archive.

  - Windows: Press `Windows + R`, type "control system", then click on **Advanced system settings** in the left

pane, then click on **Environment variables** on the window that appears.
  - Linux or macOS: Add the text `export ANDROID_HOME="/path/to/android-sdk"` to your `.bashrc` or `.zshrc` where `/path/to/android-sdk` points to the root of the SDK directories.

- Install the necessary SDK components in this folder:

  - Accept the SDK component licenses by running the following command where `android_sdk_path` is the path to the Android SDK, then answering all the prompts with `y`:

```
cmdline-tools/latest/bin/sdkmanager --sdk_root=
<android_sdk_path> --licenses
```

  - Complete setup by running the following command where `android_sdk_path` is the path to the Android SDK.

```
cmdline-tools/latest/bin/sdkmanager --sdk_root=
<android_sdk_path> "platform-tools" "build-
tools;34.0.0" "platforms;android-34" "cmdline-
tools;latest" "cmake;3.10.2.4988404" "ndk;23.2.8568313"
```

- After setting up the SDK and environment variables, be sure to **restart your terminal** to apply the changes. If you are using an IDE with an integrated terminal, you need to restart the IDE.

- Run `scons platform=android`. If this fails, go back and check the steps. If you completed the setup correctly, the NDK will begin downloading. If you are trying to compile GDExtension, you need to first compile the engine to download the NDK, then you can compile GDExtension.

Building the export templates

Godot needs three export templates for Android: the optimized "release" template (`android_release.apk`), the debug template (`android_debug.apk`), and the Gradle build template (`android_source.zip`). As Google requires all APKs to include ARMv8 (64-bit) libraries since August 2019, the commands below build templates containing both ARMv7 and ARMv8 libraries.

Compiling the standard export templates is done by calling SCons from the Godot root directory with the following arguments:

- Release template (used when exporting with **Debugging Enabled** unchecked)

```
scons platform=android target=template_release arch=arm32
scons platform=android target=template_release arch=arm64
generate_apk=yes
```

- Debug template (used when exporting with **Debugging Enabled** checked)

```
scons platform=android target=template_debug arch=arm32
scons platform=android target=template_debug arch=arm64
generate_apk=yes
```

- (**Optional**) Dev template (used when troubleshooting)

```
scons platform=android target=template_debug arch=arm32
dev_build=yes
scons platform=android target=template_debug arch=arm64
dev_build=yes generate_apk=yes
```

The resulting templates will be located under the `bin` directory:

- `bin/android_release.apk` for the release template
- `bin/android_debug.apk` for the debug template
- `bin/android_dev.apk` for the dev template
- `bin/android_source.zip` for the Gradle build template

Note

- If you are changing the list of architectures you're building, remember to add `generate_apk=yes` to the *last* architecture you're building, so that the template files are generated after the build.
- To include debug symbols in the generated templates, add the `debug_symbols=yes` parameter to the SCons command.

See also

If you want to enable Vulkan validation layers, see Vulkan validation layers on Android.

Adding support for x86 devices

If you also want to include support for x86 and x86_64 devices, run the SCons command a third and fourth time with the `arch=x86_32`, and `arch=x86_64` arguments before building the APK with Gradle. For example, for the release template:

```
scons platform=android target=template_release arch=arm32
scons platform=android target=template_release arch=arm64
scons platform=android target=template_release arch=x86_32
scons platform=android target=template_release arch=x86_64
generate_apk=yes
```

This will create template binaries that works on all platforms. The final binary size of exported projects will depend on the platforms you choose to support when exporting; in other words, unused platforms will be removed from the binary.

Cleaning the generated export templates

You can use the following commands to remove the generated export templates:

```
cd platform/android/java
# On Windows
.\gradlew clean
# On Linux and macOS
./gradlew clean
```

Using the export templates

Godot needs release and debug binaries that were compiled against the same version/commit as the editor. If you are using official binaries for the editor, make sure to install the matching export templates, or build your own from the same version.

When exporting your game, Godot uses the templates as a base, and updates their content as needed.

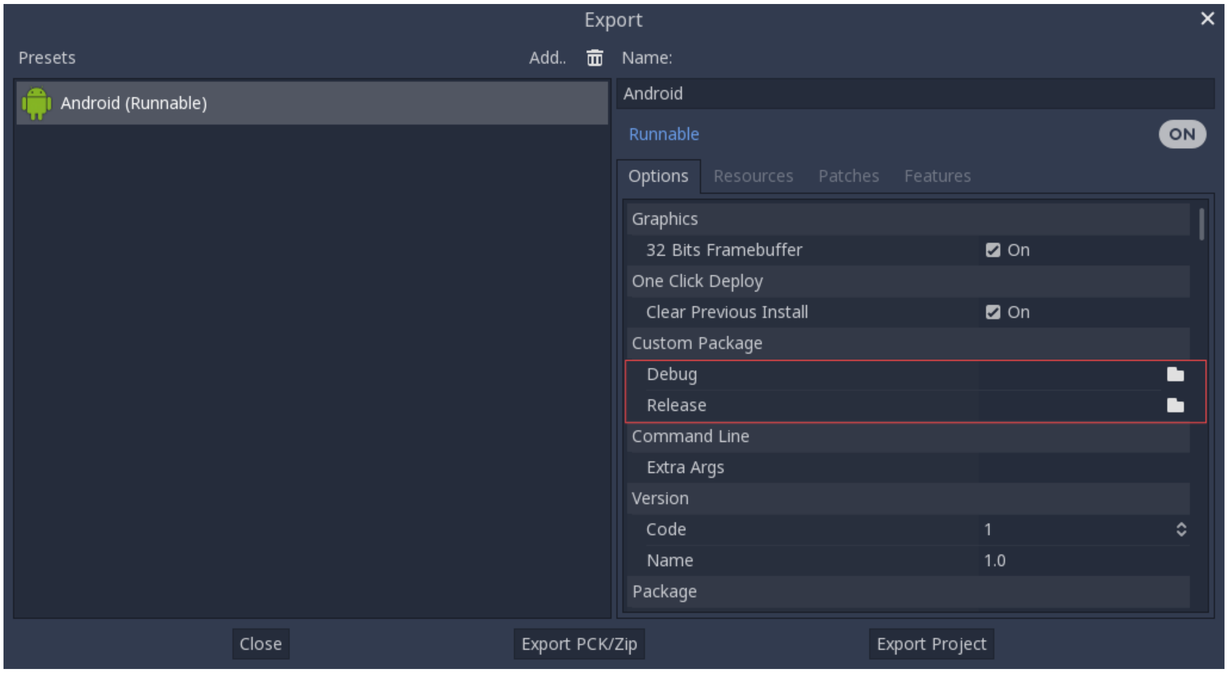
Installing the templates

The newly-compiled templates (`android_debug.apk` , `android_release.apk`, and `android_source.zip`) must be copied to Godot's templates folder with their respective names. The templates folder can be located in:

- Windows: `%APPDATA%\Godot\export_templates\<version>\`
- Linux: `$HOME/.local/share/godot/export_templates/<version>/`
- macOS: `$HOME/Library/Application Support/Godot/export_templates/<version>/`

`<version>` is of the form `major.minor[.patch].status` using values from `version.py` in your Godot source repository (e.g. `4.1.3.stable` or `4.2.dev`). You also need to write this same version string to a `version.txt` file located next to your export templates.

However, if you are writing your custom modules or custom C++ code, you might instead want to configure your template binaries as custom export templates here:


Export
Presets
Add..
Android (Runnable)
Options
Resources
Patches
Features
Close
Export PCK/Zip
Export Project

Name: Android

Runnable ON

Graphics

32 Bits Framebuffer ☑ On

One Click Deploy

Clear Previous Install ☑ On

Custom Package

Debug

Release

Command Line

Extra Args

Version

Code 1

Name 1.0

Package

You don't even need to copy them, you can just reference the resulting file in the `bin\` directory of your Godot source folder, so that the next time you build you will automatically have the custom templates referenced.

Building the Godot editor

Compiling the editor is done by calling SCons from the Godot root directory with the following arguments:

```
scons platform=android arch=arm32 production=yes target=editor
scons platform=android arch=arm64 production=yes target=editor
scons platform=android arch=x86_32 production=yes target=editor
scons platform=android arch=x86_64 production=yes target=editor
generate_apk=yes
```

- You can add the `dev_build=yes` parameter to generate a dev build of the Godot editor.
- You can add the `debug_symbols=yes` parameter to include the debug symbols in the generated build.
- You can skip certain architectures depending on your target device to speed up compilation.

Remember to add `generate_apk=yes` to the *last* architecture you're building, so that binaries are generated after the build.

The resulting binaries will be located under `bin/android_editor_builds/`.

Removing the Editor binaries

You can use the following commands to remove the generated editor binaries:

```
 cd platform/android/java
 # On Windows
.\gradlew clean
# On Linux and macOS
./gradlew clean
```

Installing the Godot editor APK

With an Android device with Developer Options enabled, connect the Android device to your computer via its charging cable to a USB/USB-C port. Open up a Terminal/Command Prompt and run the following commands from the root directory with the following arguments:

```
adb install ./bin/android_editor_builds/android_editor-
release.apk
```

Troubleshooting

Platform doesn't appear in SCons

Double-check that you've set the `ANDROID_HOME` environment variable. This is required for the platform to appear in SCons' list of detected platforms. See Setting up the buildsystem for more information.

Application not installed

Android might complain the application is not correctly installed. If so:

- Check that the debug keystore is properly generated.
- Check that the jarsigner executable is from JDK 8.

If it still fails, open a command line and run logcat [https://developer.android.com/studio/command-line/logcat]:

```
adb logcat
```

Then check the output while the application is installed; the error message should be presented there. Seek assistance if you can't figure it out.

Application exits immediately

If the application runs but exits immediately, this might be due to one of the following reasons:

- Make sure to use export templates that match your editor version; if you use a new Godot version, you *have* to update the templates too.

- `libgodot_android.so` is not in `libs/<arch>/` where `<arch>` is the device's architecture.
- The device's architecture does not match the exported one(s). Make sure your templates were built for that device's architecture, and that the export settings included support for that architecture.

In any case, `adb logcat` should also show the cause of the error.

Compiling for iOS

See also

This page describes how to compile iOS export template binaries from source. If you're looking to export your project to iOS instead, read Exporting for iOS.

Requirements

- Python 3.6+ [https://www.python.org/downloads/macos/].
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system.
- Xcode [https://apps.apple.com/us/app/xcode/id497799835].
    - Launch Xcode once and install iOS support. If you have already launched Xcode and need to install iOS support, go to *Xcode -> Settings... -> Platforms*.
    - Go to *Xcode -> Settings... -> Locations -> Command Line Tools* and select an installed version. Even if one is already selected, re-select it.

If you are building the `master` branch:

- Download and follow README instructions to build a static `.xcframework` from the MoltenVK SDK [https://github.com/KhronosGroup/MoltenVK#fetching-moltenvk-source-code].

Note

If you have Homebrew [https://brew.sh/] installed, you can easily install SCons using the following command:

```
brew install scons
```

Installing Homebrew will also fetch the Command Line Tools for Xcode automatically if you don't have them already.

Similarly, if you have MacPorts [https://www.macports.org/] installed, you can easily install SCons using the following command:

```
sudo port install scons
```

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Compiling

Open a Terminal, go to the root dir of the engine source code and type:

```
scons platform=ios target=template_debug
```

for a debug build, or:

```
scons platform=ios target=template_release
```

for a release build (check `platform/ios/detect.py` for the compiler flags used for each configuration).

Alternatively, you can run

```
scons platform=ios target=template_debug ios_simulator=yes
arch=x86_64
```

```
scons platform=ios target=template_debug ios_simulator=yes
arch=arm64
```

for a Simulator libraries.

To create an Xcode project like in the official builds, you need to use the template located in `misc/dist/ios_xcode`. The release and debug libraries should be placed in `libgodot.ios.debug.xcframework` and `libgodot.ios.release.xcframework` respectively.

```
cp -r misc/dist/ios_xcode .

cp libgodot.ios.template_debug.arm64.a
ios_xcode/libgodot.ios.debug.xcframework/ios-arm64/libgodot.a
lipo -create libgodot.ios.template_debug.arm64.simulator.a
libgodot.ios.template_debug.x86_64.simulator.a -output
ios_xcode/libgodot.ios.debug.xcframework/ios-arm64_x86_64-
simulator/libgodot.a

cp libgodot.ios.template_release.arm64.a
ios_xcode/libgodot.ios.release.xcframework/ios-arm64/libgodot.a
lipo -create libgodot.ios.template_release.arm64.simulator.a
libgodot.ios.template_release.x86_64.simulator.a -output
ios_xcode/libgodot.ios.release.xcframework/ios-arm64_x86_64-
simulator/libgodot.a
```

The MoltenVK static `.xcframework` folder must also be placed in the `ios_xcode` folder once it has been created.

Run

To run on a device or simulator, follow these instructions: Exporting for iOS.

Cross-compiling for iOS on Linux

The procedure for this is somewhat complex and requires a lot of steps, but once you have the environment properly configured you can compile Godot for iOS anytime you want.

Disclaimer

While it is possible to compile for iOS on a Linux environment, Apple is very restrictive about the tools to be used (especially hardware-wise), allowing pretty much only their products to be used for development. So this is **not official**. However, in 2010 Apple said they relaxed some of the App Store review guidelines [https://developer.apple.com/app-store/review/guidelines/] to allow any tool to be used, as long as the resulting binary does not download any code, which means it should be OK to use the procedure described here and cross-compiling the binary.

Requirements

- XCode with the iOS SDK [https://developer.apple.com/download/all/?q=Xcode] (you must be logged into an Apple ID to download Xcode).
- Clang >= 3.5 [https://clang.llvm.org] for your development machine installed and in the `PATH`. It has to be version >= 3.5 to target `arm64` architecture.
- xar [https://mackyle.github.io/xar/] and pbzx [https://github.com/NiklasRosenstein/pbzx] (required to extract the `.xip` archive Xcode comes in).

  - For building xar and pbzx, you may want to follow this guide [https://gist.github.com/phracker/1944ce190e01963c550566b749bd2b54].
- cctools-port [https://github.com/tpoechtrager/cctools-port] for the needed build tools. The procedure for building is quite peculiar and is described below.
  - This also has some extra dependencies: automake, autogen, libtool.

Configuring the environment

Preparing the SDK

Extract the Xcode `.xip` file you downloaded from Apple's developer website:

```
mkdir xcode
xar -xf /path/to/Xcode_X.x.xip -C xcode
pbzx -n Content | cpio -i

[...]
######### Blocks
```

Note that for the commands below, you will need to replace the version (`x.x`) with whatever iOS SDK version you're using. If you don't know your iPhone SDK version, you can see the JSON file inside of `Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs`.

Extract the iOS SDK:

```
export IOS_SDK_VERSION="x.x"
mkdir -p iPhoneSDK/iPhoneOS${IOS_SDK_VERSION}.sdk
cp -r
```

```
xcode/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/SDKs/iPhoneOS.sdk/*
iPhoneSDK/iPhoneOS${IOS_SDK_VERSION}.sdk
cp -r
xcode/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xcto
olchain/usr/include/c++/*
iPhoneSDK/iPhoneOS${IOS_SDK_VERSION}.sdk/usr/include/c++
fusermount -u xcode
```

Pack the SDK so that cctools can use it:

```
cd iPhoneSDK
tar -cf - * | xz -9 -c - >
iPhoneOS${IOS_SDK_VERSION}.sdk.tar.xz
```

Toolchain

Build cctools:

```
git clone https://github.com/tpoechtrager/cctools-port.git
cd cctools-port/usage_examples/ios_toolchain
./build.sh /path/iPhoneOS${IOS_SDK_VERSION}.sdk.tar.xz arm64
```

Copy the tools to a nicer place. Note that the SCons scripts for building will look under `usr/bin` inside the directory you provide for the toolchain binaries, so you must copy to such subdirectory, akin to the following commands:

```
mkdir -p "$HOME/iostoolchain/usr"
cp -r target/bin "$HOME/iostoolchain/usr/"
```

Now you should have the iOS toolchain binaries in `$HOME/iostoolchain/usr/bin`.

Compiling Godot for iPhone

Once you've done the above steps, you should keep two things in your environment: the built toolchain and the iPhoneOS SDK directory. Those can stay anywhere you want since you have to provide their paths to the SCons build command.

For the iPhone platform to be detected, you need the `OSXCROSS_IOS` environment variable defined to anything.

```
export OSXCROSS_IOS="anything"
```

Now you can compile for iPhone using SCons like the standard Godot way, with some additional arguments to provide the correct paths:

```
scons platform=ios arch=arm64 target=template_release
IOS_SDK_PATH="/path/to/iPhoneSDK"
IOS_TOOLCHAIN_PATH="/path/to/iostoolchain" ios_triple="arm-
apple-darwin11-"
```

Compiling for the Web

See also

This page describes how to compile HTML5 editor and export template binaries from source. If you're looking to export your project to HTML5 instead, read Exporting for the Web.

Requirements

To compile export templates for the Web, the following is required:

- Emscripten 3.1.39+ [https://emscripten.org].
- Python 3.6+ [https://www.python.org/].
- SCons 3.1.2+ [https://scons.org/pages/download.html] build system.

See also

To get the Godot source code for compiling, see Getting the source.

For a general overview of SCons usage for Godot, see Introduction to the buildsystem.

Note

Emscripten 3.1.39+ is recommended, but older 3.x versions are known to work.

Please note that the minimum requirement for GDExtension support is 3.1.14.

Building export templates

Before starting, confirm that `emcc` is available in your PATH. This is usually configured by the Emscripten SDK, e.g. when invoking `emsdk activate` and `source ./emsdk_env.sh`/`emsdk_env.bat`.

Open a terminal and navigate to the root directory of the engine source code. Then instruct SCons to build the Web platform. Specify `target` as either `template_release` for a release build or `template_debug` for a debug build:

```
scons platform=web target=template_release
scons platform=web target=template_debug
```

By default, the JavaScriptBridge singleton will be built into the engine. Official export templates also have the JavaScript singleton enabled. Since `eval()` calls can be a security concern, the `javascript_eval` option can be used to build without the singleton:

```
scons platform=web target=template_release javascript_eval=no
scons platform=web target=template_debug javascript_eval=no
```

By default, WebWorker threads support is enabled. To disable it and only use a single thread, the `threads` option can be used to build the web template without threads support:

```
scons platform=web target=template_release threads=no
scons platform=web target=template_debug threads=no
```

The engine will now be compiled to WebAssembly by Emscripten. Once finished, the resulting file will be placed in the `bin` subdirectory. Its name is `godot.web.template_release.wasm32.zip` for release or `godot.web.template_debug.wasm32.zip` for debug.

Finally, rename the zip archive to `web_release.zip` for the release template:

```
mv bin/godot.web.template_release.wasm32.zip
bin/web_release.zip
```

And `web_debug.zip` for the debug template:

```
mv bin/godot.web.template_debug.wasm32.zip bin/web_debug.zip
```

GDExtension

The default export templates do not include GDExtension support for performance and compatibility reasons. See the export page for more info.

You can build the export templates using the option `dlink_enabled=yes` to enable GDExtension support:

```
scons platform=web dlink_enabled=yes target=template_release
scons platform=web dlink_enabled=yes target=template_debug
```

Once finished, the resulting file will be placed in the `bin` subdirectory. Its name will have `_dlink` added.

Finally, rename the zip archives to `web_dlink_release.zip` and `web_dlink_release.zip` for the release template:

```
mv bin/godot.web.template_release.wasm32.dlink.zip
bin/web_dlink_release.zip
mv bin/godot.web.template_debug.wasm32.dlink.zip
bin/web_dlink_debug.zip
```

Building the editor

It is also possible to build a version of the Godot editor that can run in the browser. The editor version is not recommended over the native build. You can build the editor with:

```
scons platform=web target=editor
```

Once finished, the resulting file will be placed in the `bin` subdirectory. Its name will be `godot.web.editor.wasm32.zip`. You can upload the zip content to your web server and visit it with your browser to use the editor.

Refer to the export page for the web server requirements.

Tip

The Godot repository includes a Python script to host a local web server [https://raw.githubusercontent.com/godotengine/godot/master/platform/web/serve.py]. This can be used to test the web editor locally.

After compiling the editor, extract the ZIP archive that was created in the `bin/` folder, then run the following command in the Godot repository root:

```
# You may need to replace `python` with `python3` on some platforms.
python platform/web/serve.py
```

This will serve the contents of the `bin/` folder and open the default web browser automatically. In the page that opens, access `godot.tools.html` and you should be able to test the web editor this way.

Note that for production use cases, this Python-based web server should not be used. Instead, you should use an established web server such as Apache or nginx.

Compiling with .NET

Requirements

- .NET SDK 8.0+ [https://dotnet.microsoft.com/download]

  You can use `dotnet --info` to check which .NET SDK versions are installed.

Enable the .NET module

Note

C# support for Godot has historically used the Mono [https://www.mono-project.com/] runtime instead of the .NET Runtime [https://github.com/dotnet/runtime] and internally many things are still named `mono` instead of `dotnet` or otherwise referred to as `mono`.

By default, the .NET module is disabled when building. To enable it, add the option `module_mono_enabled=yes` to the SCons command line, while otherwise following the instructions for building the desired Godot binaries.

Generate the glue

Parts of the sources of the managed libraries are generated from the ClassDB. These source files must be generated before building the managed libraries. They can be generated by any .NET-enabled Godot editor binary by running it with the parameters `--headless --`

`generate-mono-glue` followed by the path to an output directory. This path must be `modules/mono/glue` in the Godot directory:

```
<godot_binary> --headless --generate-mono-glue
modules/mono/glue
```

This command will tell Godot to generate the C# bindings for the Godot API at `modules/mono/glue/GodotSharp/GodotSharp/Generated`, and the C# bindings for the editor tools at `modules/mono/glue/GodotSharp/GodotSharpEditor/Generated`. Once these files are generated, you can build Godot's managed libraries for all the desired targets without having to repeat this process.

`<godot_binary>` refers to the editor binary you compiled with the .NET module enabled. Its exact name will differ based on your system and configuration, but should be of the form `bin/godot.<platform>.editor.<arch>.mono`, e.g. `bin/godot.linuxbsd.editor.x86_64.mono` or `bin/godot.windows.editor.x86_32.mono.exe`. Be especially aware of the **.mono** suffix! If you've previously compiled Godot without .NET support, you might have similarly named binaries without this suffix. These binaries can't be used to generate the .NET glue.

Note

The glue sources must be regenerated every time the ClassDB-registered API changes. That is, for example, when a new method is registered to the scripting API or one of the parameters of such a method changes. Godot will print an error at startup if there is an API mismatch between ClassDB and the glue sources.

Building the managed libraries

Once you have generated the .NET glue, you can build the managed libraries with the `build_assemblies.py` script:

```
./modules/mono/build_scripts/build_assemblies.py --godot-
output-dir=./bin
```

If everything went well, the `GodotSharp` directory, containing the managed libraries, should have been created in the `bin` directory.

Note

By default, all development builds share a version number, which can cause some issues with caching of the NuGet packages. To solve this issue either use `GODOT_VERSION_STATUS` to give every build a unique version or delete `GodotNuGetFallbackFolder` after every build to clear the package cache.

Unlike "classical" Godot builds, when building with the .NET module enabled (and depending on the target platform), a data directory may be created both for the editor and for exported projects. This directory is important for proper functioning and must be distributed together with Godot. More details about this directory in Data directory.

Build Platform

Provide the `--godot-platform=<platform>` argument to control for which platform specific the libraries are built. Omit this argument to build for the current system.

This currently only controls the inclusion of the support for Visual Studio as an external editor, the libraries are otherwise identical.

NuGet packages

The API assemblies, source generators, and custom MSBuild project SDK are distributed as NuGet packages. This is all transparent to the user, but it can make things complicated during development.

In order to use Godot with a development version of those packages, a local NuGet source must be created where MSBuild can find them.

First, pick a location for the local NuGet source. If you don't have a preference, create an empty directory at one of these recommended locations:

- On Windows, `C:\Users\<username>\MyLocalNugetSource`
- On Linux, *BSD, etc., `~/MyLocalNugetSource`

This path is referred to later as `<my_local_source>`.

After picking a directory, run this .NET CLI command to configure NuGet to use your local source:

```
dotnet nuget add source <my_local_source> --name
MyLocalNugetSource
```

When you run the `build_assemblies.py` script, pass `<my_local_source>` to the `--push-nupkgs-local` option:

```
./modules/mono/build_scripts/build_assemblies.py --godot-
output-dir ./bin --push-nupkgs-local <my_local_source>
```

This option ensures the packages will be added to the specified local NuGet source and that conflicting versions of the package are removed from the NuGet cache. It's recommended to always use this option when building the C# solutions during development to avoid mistakes.

Double Precision Support (REAL_T_IS_DOUBLE)

When building Godot with double precision support, i.e. the `precision=double` argument for scons, the managed libraries must be adjusted to match by passing the `--precision=double` argument:

```
./modules/mono/build_scripts/build_assemblies.py --godot-
output-dir ./bin --push-nupkgs-local <my_local_source> --
precision=double
```

Examples

Example (Windows)

```
# Build editor binary
scons platform=windows target=editor module_mono_enabled=yes
# Build export templates
scons platform=windows target=template_debug
module_mono_enabled=yes
scons platform=windows target=template_release
module_mono_enabled=yes

# Generate glue sources
bin/godot.windows.editor.x86_64.mono --headless --generate-
mono-glue modules/mono/glue
# Build .NET assemblies
./modules/mono/build_scripts/build_assemblies.py --godot-
output-dir=./bin --godot-platform=windows
```

Example (Linux, *BSD)

```
# Build editor binary
scons platform=linuxbsd target=editor module_mono_enabled=yes
# Build export templates
scons platform=linuxbsd target=template_debug
module_mono_enabled=yes
scons platform=linuxbsd target=template_release
module_mono_enabled=yes

# Generate glue sources
```

```
bin/godot.linuxbsd.editor.x86_64.mono --headless --generate-mono-glue modules/mono/glue
# Generate binaries
./modules/mono/build_scripts/build_assemblies.py --godot-output-dir=./bin --godot-platform=linuxbsd
```

Data directory

The data directory is a dependency for Godot binaries built with the .NET module enabled. It contains important files for the correct functioning of Godot. It must be distributed together with the Godot executable.

Editor

The name of the data directory for the Godot editor will always be `GodotSharp`. This directory contains an `Api` subdirectory with the Godot API assemblies and a `Tools` subdirectory with the tools required by the editor, like the `GodotTools` assemblies and its dependencies.

On macOS, if the Godot editor is distributed as a bundle, the `GodotSharp` directory may be placed in the `<bundle_name>.app/Contents/Resources/` directory inside the bundle.

Export templates

The data directory for exported projects is generated by the editor during the export. It is named `data_<APPNAME>_<ARCH>`, where `<APPNAME>` is the application name as specified in the project setting `application/config/name` and `<ARCH>` is the current architecture of the export.

In the case of multi-architecture exports multiple such data directories will be generated.

Command-line options

The following is the list of command-line options available when building with the .NET module:

- **module_mono_enabled**=yes | **no**
    - Build Godot with the .NET module enabled.

Compiling with PCK encryption key

The export dialog gives you the option to encrypt your PCK file with a 256-bit AES key when releasing your project. This will make sure your scenes, scripts and other resources are not stored in plain text and can not easily be ripped by some script kiddie.

Of course, the key needs to be stored in the binary, but if it's compiled, optimized and without symbols, it would take some effort to find it.

For this to work, you need to build the export templates from source, with that same key.

Warning

This will **not** work if you use official, precompiled export templates. It is absolutely **required** to compile your own export templates to use PCK encryption.

Warning

By default, Android exports store assets directly in the APK file and aren't affected by PCK encryption. To use PCK encryption on Android, enable **APK expansion** in the export options.

Step by step

1. Generate a 256-bit AES key in hexadecimal format. You can use the aes-256-cbc variant from this service [https://asecuritysite.com/encryption/keygen].

   Alternatively, you can generate it yourself using OpenSSL [https://www.openssl.org/] command-line tools:

   ```
   openssl rand -hex 32 > godot.gdkey
   ```

   The output in `godot.gdkey` should be similar to:

   ```
   # NOTE: Do not use the key below! Generate your own key
   instead.
   aeb1bc56aaf580cc31784e9c41551e9ed976ecba10d315db591e749f3f6
   4890f
   ```

   You can generate the key without redirecting the output to a file, but that way you can minimize the risk of exposing the key.

2. Set this key as environment variable in the console that you will use to compile Godot, like this:

   Linux/macOS

   ```
   export SCRIPT_AES256_ENCRYPTION_KEY="your_generated_key"
   ```

   Windows (cmd)

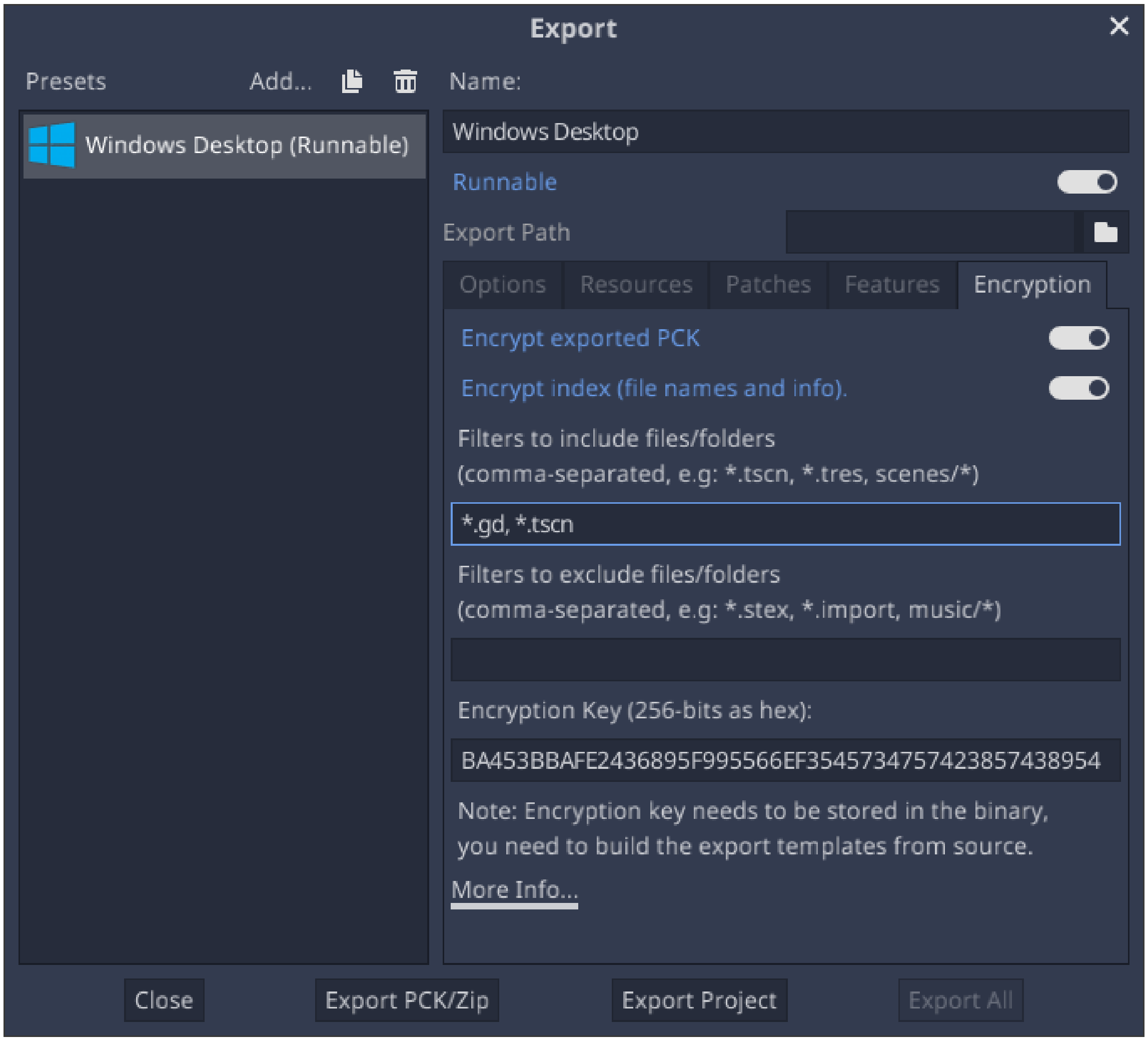
   ```
   set SCRIPT_AES256_ENCRYPTION_KEY=your_generated_key
   ```

   Windows (PowerShell)

   ```
   $env:SCRIPT_AES256_ENCRYPTION_KEY="your_generated_key"
   ```

3. Compile Godot export templates and set them as custom export templates in the export preset options.

4. Set the encryption key in the **Encryption** tab of the export preset:


Export
Presets
Add...
Options
Resources
Patches
Features
Encryption
More Info...
Close
Export PCK/Zip
Export Project
Export All

Windows Desktop (Runnable)

Name: Windows Desktop

Runnable ●

Export Path

Encrypt exported PCK ●

Encrypt index (file names and info). ●

Filters to include files/folders
(comma-separated, e.g: *.tscn, *.tres, scenes/*)

*.gd, *.tscn

Filters to exclude files/folders
(comma-separated, e.g: *.stex, *.import, music/*)

Encryption Key (256-bits as hex):

BA453BBAFE2436895F995566EF3545734757423857438954

Note: Encryption key needs to be stored in the binary,
you need to build the export templates from source.

5. Add filters for the files/folders to encrypt. **By default**, include filters are empty and **nothing will be encrypted**.
6. Export the project. The project should run with the files encrypted now.

Troubleshooting

If you get an error like below, it means the key wasn't properly included in your Godot build. Godot is encrypting PCK file during export, but can't read it at runtime.

```
ERROR: open_and_parse: Condition "String::md5(md5.digest) !=
String::md5(md5d)" is true. Returning: ERR_FILE_CORRUPT
   At: core/io/file_access_encrypted.cpp:103
```

Optimizing a build for size

Rationale

Sometimes, it is desired to optimize a build for size rather than speed. This means not compiling unused functions from the engine, as well as using specific compiler flags to aid on decreasing build size. Common situations include creating builds for mobile and Web platforms.

This tutorial aims to give an overview on different methods to create a smaller binary. Before continuing, it is recommended to read the previous tutorials on compiling Godot for each platform.

The options below are listed from the most important (greatest size savings) to the least important (lowest size savings).

Stripping binaries

- **Space savings:** Very high
- **Difficulty:** Easy
- **Performed in official builds:** Yes

If you build Windows (MinGW), Linux or macOS binaries from source, remember to strip debug symbols from binaries by installing the `strip` package from your distribution then running:

```
strip path/to/godot.binary
```

On Windows, `strip.exe` is included in most MinGW toolchain setups.

This will reduce the size of compiled binaries by a factor between 5× and 10×. The downside is that crash backtraces will no longer provide accurate information (which is useful for troubleshooting the cause of a crash). C++ profilers will also no longer be able to display function names (this does not affect the built-in GDScript profiler).

Note

The above command will not work on Windows binaries compiled with MSVC and platforms such as Android and Web. Instead, pass `debug_symbols=no` on the SCons command line when compiling.

Compiling with link-time optimization

- **Space savings:** High
- **Difficulty:** Easy
- **Performed in official builds:** Yes

Enabling link-time optimization produces more efficient binaries, both in terms of performance and file size. It works by eliminating duplicate template functions and unused code. It can currently be used with the GCC and MSVC compilers:

```
scons target=template_release lto=full
```

Linking becomes much slower and more RAM-consuming with this option, so it should be used only for release builds:

- When compiling the `master` branch, you need to have at least 8 GB of RAM available for successful linking with LTO enabled.
- When compiling the `3.x` branch, you need to have at least 6 GB of RAM available for successful linking with LTO enabled.

Optimizing for size instead of speed

- **Space savings:** High
- **Difficulty:** Easy
- **Performed in official builds:** Yes, but only for web builds

Godot 3.1 onwards allows compiling using size optimizations (instead of speed). To enable this, set the `optimize` flag to `size`:

```
scons target=template_release optimize=size
```

Some platforms such as WebAssembly already use this mode by default.

Disabling advanced text server

- **Space savings:** High
- **Difficulty:** Easy
- **Performed in official builds:** No

By default, Godot uses an advanced text server with the support for the following features:

- Right-to-left typesetting and complex scripts, required to write languages such as Arabic and Hebrew.
- Font ligatures and OpenType features (such as small capitals, fractions and slashed zero).

Godot provides a fallback text server that isn't compiled by default. This text server can be used as a lightweight alternative to the default advanced text server:

```
scons target=template_release module_text_server_adv_enabled=no
module_text_server_fb_enabled=yes
```

If you only intend on supporting Latin, Greek and Cyrillic-based languages in your project, the fallback text server should suffice.

This fallback text server can also process large amounts of text more quickly than the advanced text server. This makes the fallback text server a good fit for mobile/web projects.

Note

Remember to always pass `module_text_server_fb_enabled=yes` when using `module_text_server_adv_enabled=no`. Otherwise, the compiled binary won't contain any text server, which means no text will be displayed at all when running the project.

Disabling 3D

- **Space savings:** Moderate
- **Difficulty:** Easy
- **Performed in official builds:** No

For 2D games, having the whole 3D engine available usually makes no sense. Because of this, there is a build flag to disable it:

```
scons target=template_release disable_3d=yes
```

Tools must be disabled in order to use this flag, as the editor is not designed to operate without 3D support. Without it, the binary size can be reduced by about 15%.

Note

Disabling 3D support also disables all navigation. This includes 2D navigation, as it uses the 3D navigation system internally.

Disabling advanced GUI objects

- **Space savings:** Moderate
- **Difficulty:** Easy
- **Performed in official builds:** No

Most small games don't require complex GUI controls such as Tree, ItemList, TextEdit or GraphEdit. They can be disabled using a build flag:

```
scons target=template_release disable_advanced_gui=yes
```

This is everything that will be disabled:

- FileDialog
- PopupMenu
- Tree
- TextEdit
- CodeEdit
- SyntaxHighlighter
- CodeHighlighter
- TreeItem
- OptionButton
- SpinBox
- ColorPicker
- ColorPickerButton
- RichTextlabel
- RichTextEffect
- CharFXTransform
- AcceptDialog
- ConfirmationDialog
- MarginContainer
- SubViewportContainer
- SplitContainer
- HSplitContainer
- VSplitContainer
- GraphNode
- GraphEdit

Disabling unwanted modules

- **Space savings:** Very low to moderate depending on modules
- **Difficulty:** Medium to hard depending on modules
- **Performed in official builds:** No

A lot of Godot's functions are offered as modules. You can see a list of modules with the following command:

```
scons --help
```

The list of modules that can be disabled will appear, together with all build options. If you are working on a simple 2D game, you could disable a lot of them:

```
scons target=template_release module_basis_universal_enabled=no
module_bmp_enabled=no module_camera_enabled=no
module_csg_enabled=no module_dds_enabled=no
module_enet_enabled=no module_gridmap_enabled=no
module_hdr_enabled=no module_jsonrpc_enabled=no
module_ktx_enabled=no module_mbedtls_enabled=no
module_meshoptimizer_enabled=no module_minimp3_enabled=no
module_mobile_vr_enabled=no module_msdfgen_enabled=no
module_multiplayer_enabled=no module_noise_enabled=no
module_navigation_enabled=no module_ogg_enabled=no
module_openxr_enabled=no module_raycast_enabled=no
module_regex_enabled=no module_squish_enabled=no
module_svg_enabled=no module_tga_enabled=no
module_theora_enabled=no module_tinyexr_enabled=no
module_upnp_enabled=no module_vhacd_enabled=no
module_vorbis_enabled=no module_webrtc_enabled=no
module_websocket_enabled=no module_webxr_enabled=no
module_zip_enabled=no
```

If this proves not to work for your use case, you should review the list of modules and see which ones you actually still need for your game (e.g. you might want to keep networking-related modules, regex support, `minimp3`/`ogg`/`vorbis` to play music, or `theora` to play videos).

Alternatively, you can supply a list of disabled modules by creating `custom.py` at the root of the source, with the contents similar to the following:

```
# custom.py

module_basis_universal_enabled = "no"
module_bmp_enabled = "no"
module_camera_enabled = "no"
module_csg_enabled = "no"
module_dds_enabled = "no"
module_enet_enabled = "no"
module_gridmap_enabled = "no"
module_hdr_enabled = "no"
module_jsonrpc_enabled = "no"
module_ktx_enabled = "no"
module_mbedtls_enabled = "no"
module_meshoptimizer_enabled = "no"
module_minimp3_enabled = "no"
module_mobile_vr_enabled = "no"
module_msdfgen_enabled= "no"
module_multiplayer_enabled = "no"
module_noise_enabled = "no"
module_navigation_enabled = "no"
module_ogg_enabled = "no"
module_openxr_enabled = "no"
module_raycast_enabled = "no"
module_regex_enabled = "no"
module_squish_enabled = "no"
module_svg_enabled = "no"
module_tga_enabled = "no"
module_theora_enabled = "no"
module_tinyexr_enabled = "no"
module_upnp_enabled = "no"
module_vhacd_enabled = "no"
module_vorbis_enabled = "no"
module_webrtc_enabled = "no"
module_websocket_enabled = "no"
module_webxr_enabled = "no"
module_zip_enabled = "no"
```

See also

Overriding the build options.

Optimizing the distribution of your project

Desktop

Note

This section is only relevant when distributing the files on a desktop platform that doesn't perform its own compression or packing. As such, this advice is relevant when you distribute ZIP archives on itch.io or GitHub Releases.

Platforms like Steam already apply their own compression scheme, so you don't need to create a ZIP archive to distribute files in the first place.

As an aside, you can look into optimizing the distribution of your project itself. This can be done even without recompiling the export template.

7-Zip [https://7-zip.org/] can be used to create ZIP archives that are more efficient than usual, while remaining compatible with every ZIP extractor (including Windows' own built-in extractor). ZIP size reduction in a large project can reach dozens of megabytes compared to a typical ZIP compressor, although average savings are in the 1-5 MB range. Creating this ZIP archive will take longer than usual, but it will extract just as fast as any other ZIP archive.

When using the 7-Zip GUI, this is done by creating a ZIP archive with the Ultra compression mode. When using the command line, this is done using the following command:

```
7z a -mx9 my_project.zip folder_containing_executable_and_pck
```

Web

Enabling gzip or Brotli compression for all file types from the web export (especially the `.wasm` and `.pck`) can reduce the download size significantly, leading to faster loading times, especially on slow connections.

Creating precompressed gzip or Brotli files with a high compression level can be even more efficient, as long as the web server is configured to serve those files when they exist. When supported, Brotli should be preferred over gzip as it has a greater potential for file size reduction.

See Serving the files for instructions.

Debugging and profiling

This section contains pages that provide guidance if you're looking at the engine code trying to find an underlying issue or an optimization possibility.

- Using C++ profilers
- Using sanitizers
- Debugging on macOS
- Vulkan

Debugging the editor

When working on the Godot editor keep in mind that by default the executable will start in the Project Manager mode. Opening a project from the Project Manager spawns a new process, which stops the debugging session. To avoid that you should launch directly into the project using `-e` and `--path` launch options.

For example, using `gdb` directly, you may do this:

```
gdb godot
> run -e --path ~/myproject
```

You can also run the editor directly from your project's folder. In that case, only the `-e` option is required.

```
cd ~/myproject
gdb godot
> run -e
```

You can learn more about these launch options and other command line arguments in the command line tutorial.

If you're using a code editor or an IDE to debug Godot, check out our configuration guides, which cover the setup process for building and debugging with your particular editor.

Using C++ profilers

To optimize Godot's performance, you need to know what to optimize first. To this end, profilers are useful tools.

Note

There is a built-in GDScript profiler in the editor, but using C++ profiler may be useful in cases where the GDScript profiler is not accurate enough or is missing information due to bugs in the profiler.

Recommended profilers

- VerySleepy [http://www.codersnotes.com/sleepy/] (Windows only)
- HotSpot [https://github.com/KDAB/hotspot] (Linux only)
- Xcode Instruments [https://developer.apple.com/xcode/] (macOS only)

These profilers may not be the most powerful or flexible options, but their standalone operation and limited feature set tends to make them easier to use.

Setting up Godot

To get useful profiling information, it is **absolutely required** to use a Godot build that includes debugging symbols. Official binaries do not include debugging symbols, since these would make the download size significantly larger.

To get profiling data that best matches the production environment (but with debugging symbols), you should compile binaries with the `production=yes debug_symbols=yes` SCons options.

It is possible to run a profiler on less optimized builds (e.g. `target=template_debug` without LTO), but results will naturally be less representative of real world conditions.

Warning

Do *not* strip debugging symbols on the binaries using the `strip` command after compiling the binaries. Otherwise, you will no longer get useful profiling information when running a profiler.

Benchmarking startup/shutdown times

If you're looking into optimizing Godot's startup/shutdown performance, you can tell the profiler to use the `--quit` command line option on the Godot binary. This will exit Godot just after it finished starting. The `--quit` option works with `--editor`, `--project-manager` or `--path <path to project directory>` (which runs a project directly).

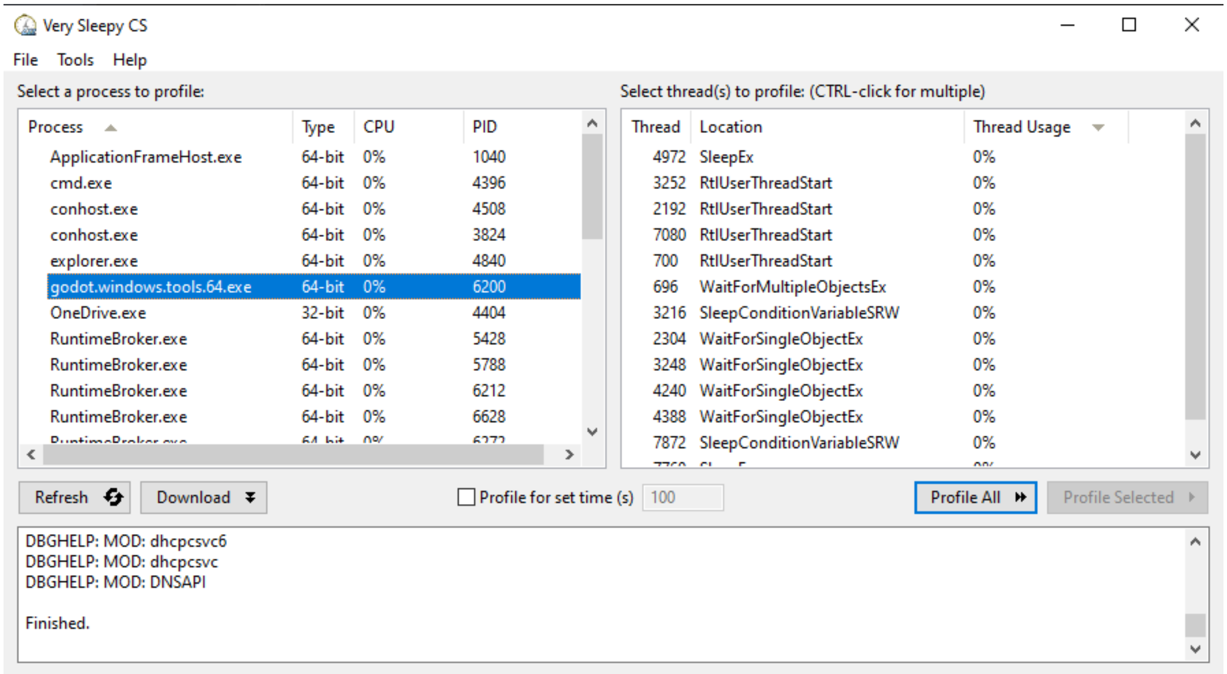
See also

See Command line tutorial for more command line arguments supported by Godot.

Profiler-specific instructions

VerySleepy

- Start the Godot editor or your project first. If you start the Project Manager, make sure to edit or run a project first. Otherwise, the profiler will not track the child process since the Project Manager will spawn a child process for every project edited or run.
- Open VerySleepy and select the Godot executable in the list of processes on the left:



Select a process to profile:

Process	Type	CPU	PID
ApplicationFrameHost.exe	64-bit	0%	1040
cmd.exe	64-bit	0%	4396
conhost.exe	64-bit	0%	4508
conhost.exe	64-bit	0%	3824
explorer.exe	64-bit	0%	4840
godot.windows.tools.64.exe	64-bit	0%	6200
OneDrive.exe	32-bit	0%	4404
RuntimeBroker.exe	64-bit	0%	5428
RuntimeBroker.exe	64-bit	0%	5788
RuntimeBroker.exe	64-bit	0%	6212
RuntimeBroker.exe	64-bit	0%	6628

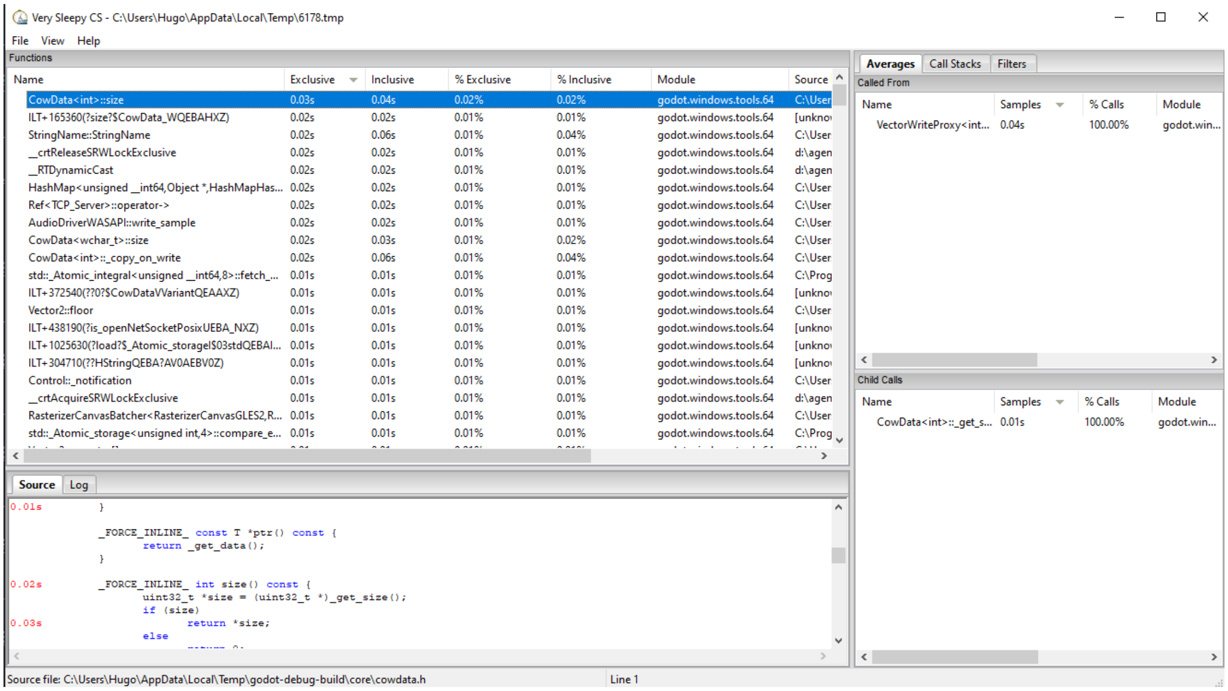
Select thread(s) to profile: (CTRL-click for multiple)

Thread	Location	Thread Usage
4972	SleepEx	0%
3252	RtlUserThreadStart	0%
2192	RtlUserThreadStart	0%
7080	RtlUserThreadStart	0%
700	RtlUserThreadStart	0%
696	WaitForMultipleObjectsEx	0%
3216	SleepConditionVariableSRW	0%
2304	WaitForSingleObjectEx	0%
3248	WaitForSingleObjectEx	0%
4240	WaitForSingleObjectEx	0%
4388	WaitForSingleObjectEx	0%
7872	SleepConditionVariableSRW	0%

```
DBGHELP: MOD: dhcpcsvc6
DBGHELP: MOD: dhcpcsvc
DBGHELP: MOD: DNSAPI

Finished.
```

- Click the **Profile All** button on the right to start profiling.
- Perform the actions you wish to profile in the editor or project. When you're done, click **Stop** (*not* Abort).
- Wait for the results window to appear.
- Once the results window appears, filter the view to remove external modules (such as the graphics driver). You can filter by module by finding a line whose **Module** matches the Godot executable name, right-clicking that line then choosing **Filter Module to <Godot executable name>** in the dropdown that appears.

- Your results window should now look something like this:



Name	Exclusive	Inclusive	% Exclusive	% Inclusive	Module	Source
CowData<int>::size	0.03s	0.04s	0.02%	0.02%	godot.windows.tools.64	C:\User
ILT+165360(?size?$CowData_WQEBAHXZ)	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	[unknov
StringName::StringName	0.02s	0.06s	0.01%	0.04%	godot.windows.tools.64	C:\User
__crtReleaseSRWLockExclusive	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	d:\agen
__RTDynamicCast	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	d:\agen
HashMap<unsigned __int64,Object *,HashMapHas...	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	C:\User
Ref<TCP_Server>::operator->	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	C:\User
AudioDriverWASAPI::write_sample	0.02s	0.02s	0.01%	0.01%	godot.windows.tools.64	C:\User
CowData<wchar_t>::size	0.02s	0.03s	0.01%	0.02%	godot.windows.tools.64	C:\User
CowData<int>::_copy_on_write	0.02s	0.06s	0.01%	0.04%	godot.windows.tools.64	C:\User
std::_Atomic_integral<unsigned __int64,8>::fetch_...	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	C:\Prog
ILT+372540(??0?$CowDataVVariantQEAAXZ)	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	[unknov
Vector2::floor	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	C:\User
ILT+438190(?is_openNetSocketPosixUEBA_NXZ)	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	[unknov
ILT+1025630(?load?$_Atomic_storageI$03stdQEBAI...	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	[unknov
ILT+304710(??HStringQEBA?AV0AEBV0Z)	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	[unknov
Control::_notification	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	C:\User
__crtAcquireSRWLockExclusive	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	d:\agen
RasterizerCanvasBatcher<RasterizerCanvasGLES2,R...	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	C:\User
std::_Atomic_storage<unsigned int,4>::compare_e...	0.01s	0.01s	0.01%	0.01%	godot.windows.tools.64	C:\Prog

```
0.01s           }

                _FORCE_INLINE_ const T *ptr() const {
                        return _get_data();
                }

0.02s           _FORCE_INLINE_ int size() const {
                        uint32_t *size = (uint32_t *)_get_size();
                        if (size)
0.03s                           return *size;
                        else
```

Source file: C:\Users\Hugo\AppData\Local\Temp\godot-debug-build\core\cowdata.h   Line 1

Called From

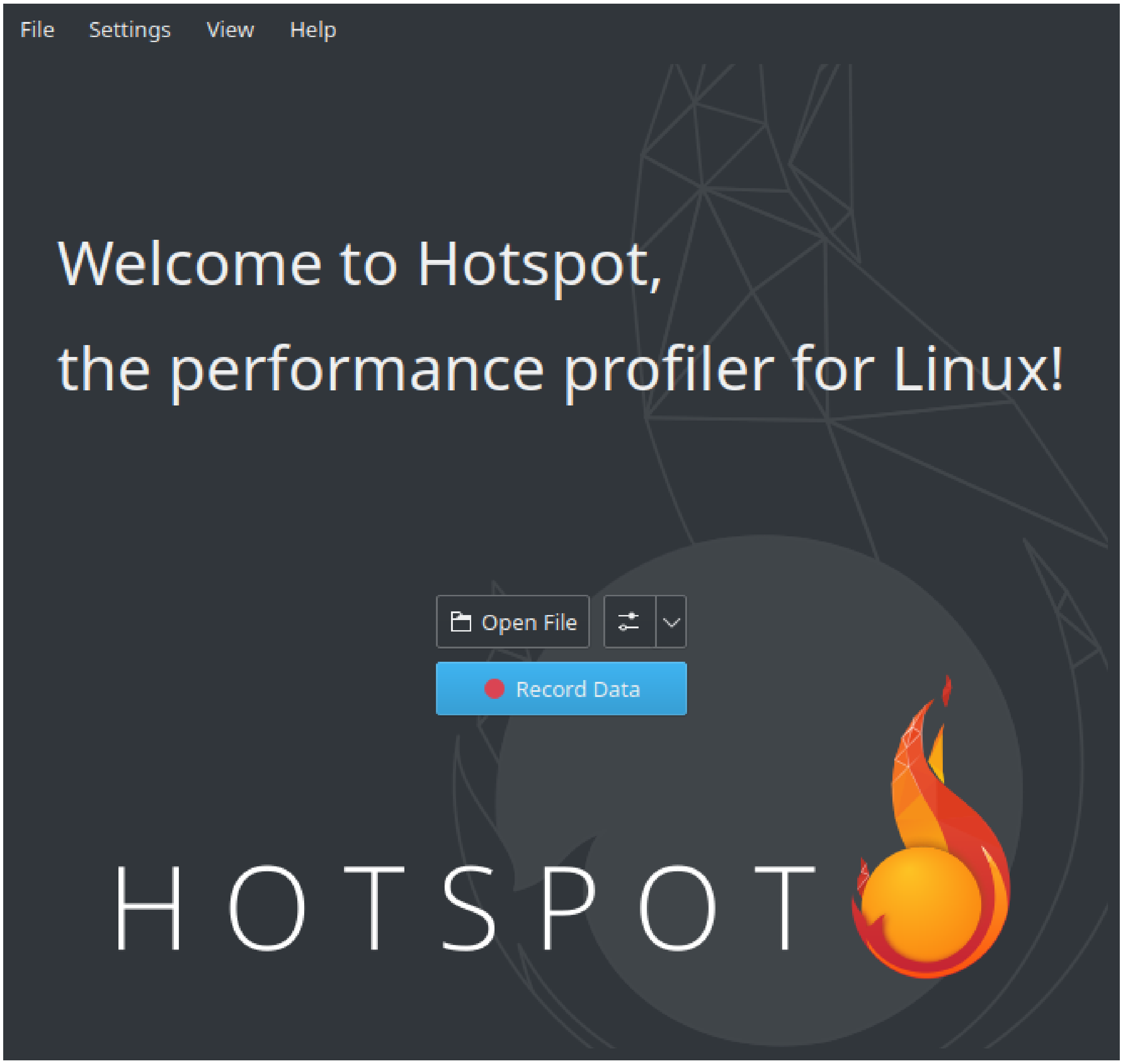
Name	Samples	% Calls	Module
VectorWriteProxy<int...	0.04s	100.00%	godot.win...

Child Calls

Name	Samples	% Calls	Module
CowData<int>::_get_s...	0.01s	100.00%	godot.win...

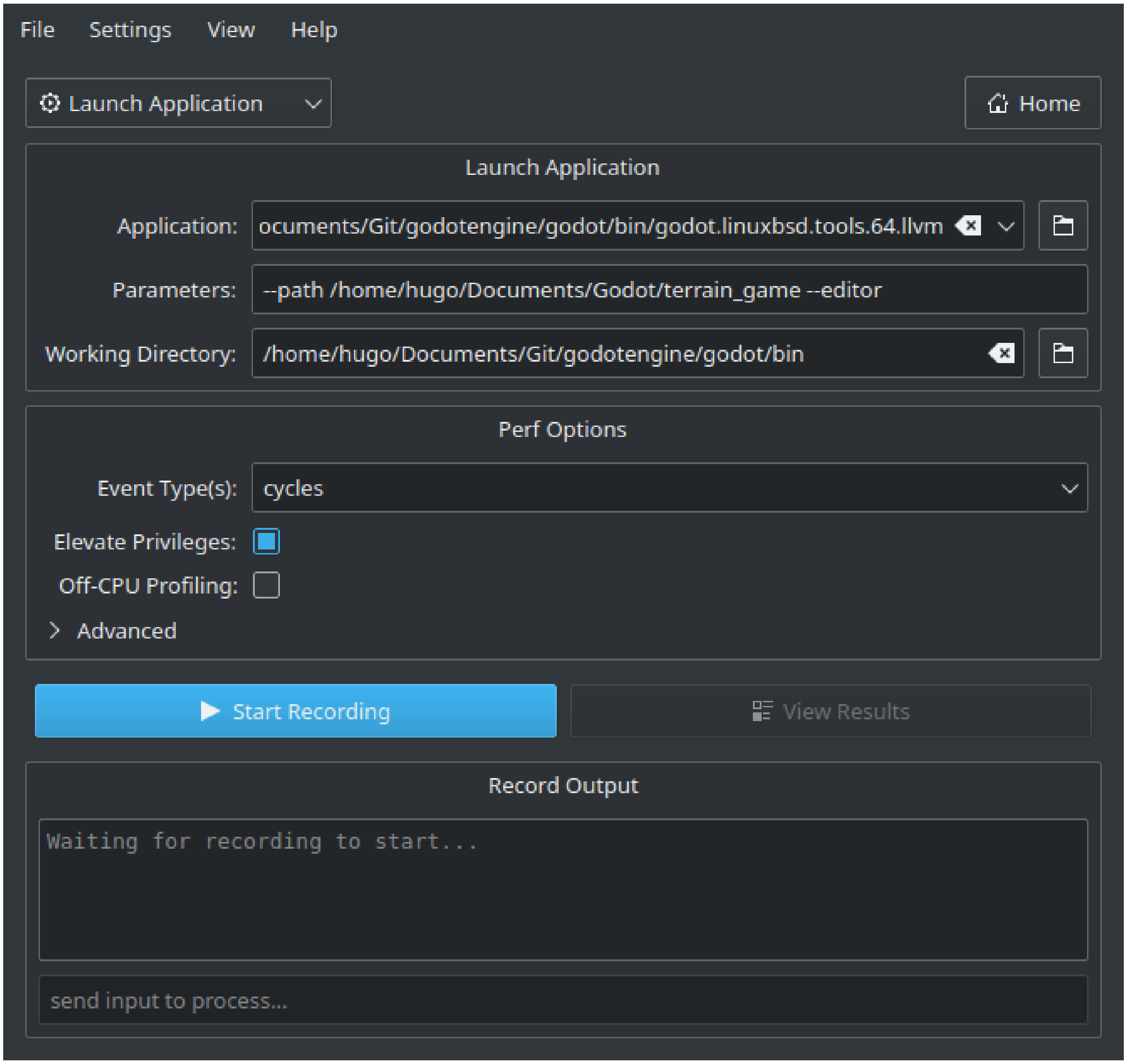
HotSpot

- Open HotSpot. Click **Record Data**:


File
Settings
View
Help
Welcome to Hotspot,
the performance profiler for Linux!
Open File
Record Data
HOTSPOT

- In the next window, specify the path to the Godot binary that includes debug symbols.
- Specify command line arguments to run a specific project, with or without the editor.
- The path to the working directory can be anything if an absolute path is used for the `--path` command line argument. Otherwise, it must be set to that the relative path to the project is valid.
- Make sure **Elevate Privileges** is checked if you have administrative privileges. While not essential for profiling Godot, this will ensure all events can be captured. Otherwise, some

events may be missing in the capture. Your settings should now look something like this:


File
Settings
View
Help
Launch Application
Home
Launch Application
Perf Options
Advanced
Start Recording
View Results
Record Output
send input to process...

Application: ocuments/Git/godotengine/godot/bin/godot.linuxbsd.tools.64.llvm

Parameters: --path /home/hugo/Documents/Godot/terrain_game --editor

Working Directory: /home/hugo/Documents/Git/godotengine/godot/bin

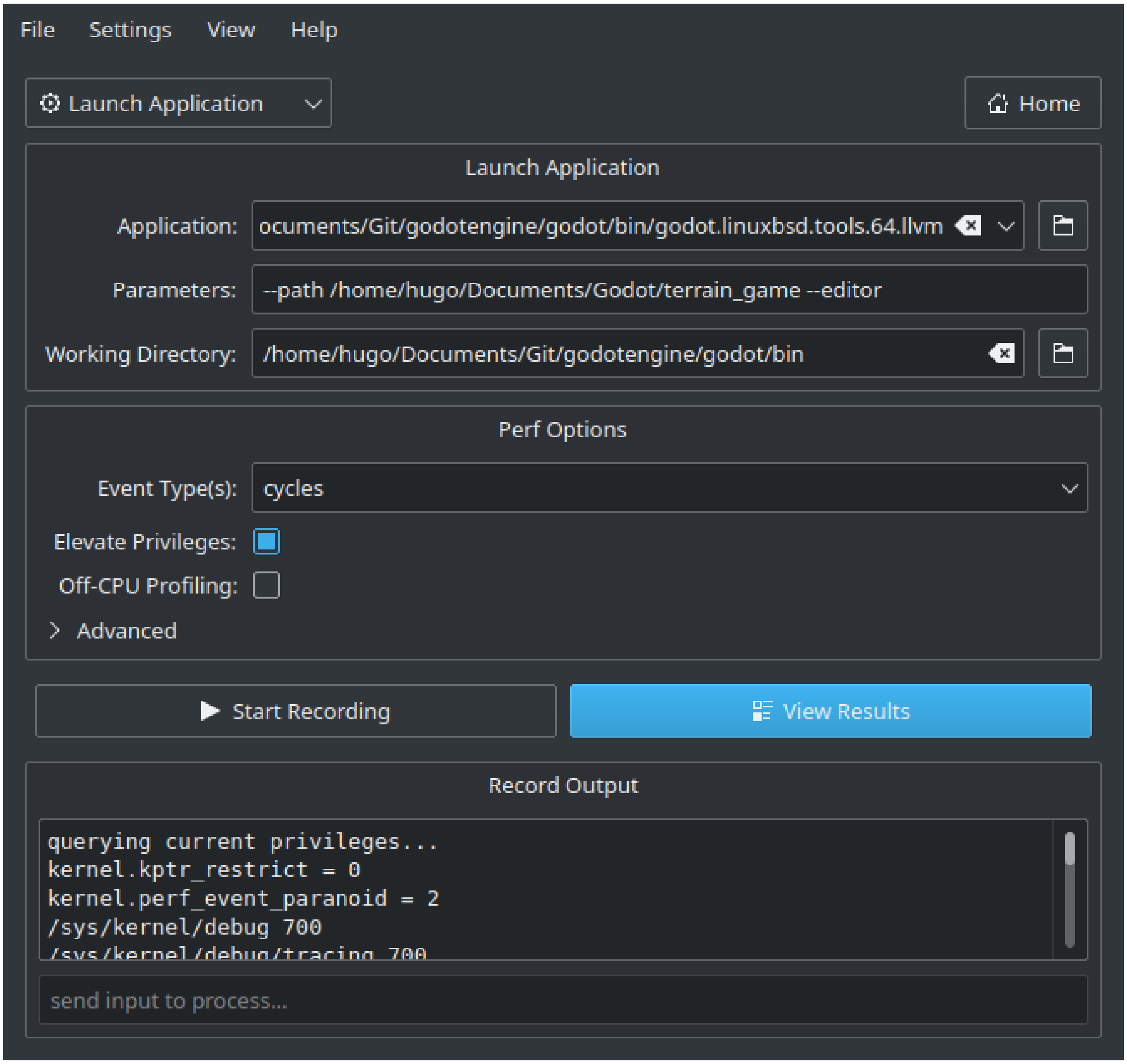
Event Type(s): cycles

Elevate Privileges: ☑

Off-CPU Profiling: ☐

```
Waiting for recording to start...
```

- Click **Start Recording** and perform the actions you wish to profile in the editor/project.
- Quit the editor/project normally or use the **Stop Profiling** button in HotSpot to stop profiling early. Stopping profiling early can result in cleaner profiles if you're not interested in the engine's quit procedure.
- Click **View Results** and wait for the profiling visualization to be generated:


File
Settings
View
Help
Launch Application
Home
Launch Application
Perf Options
Advanced
Start Recording
View Results
Record Output
send input to process...

Application: ocuments/Git/godotengine/godot/bin/godot.linuxbsd.tools.64.llvm

Parameters: --path /home/hugo/Documents/Godot/terrain_game --editor

Working Directory: /home/hugo/Documents/Git/godotengine/godot/bin

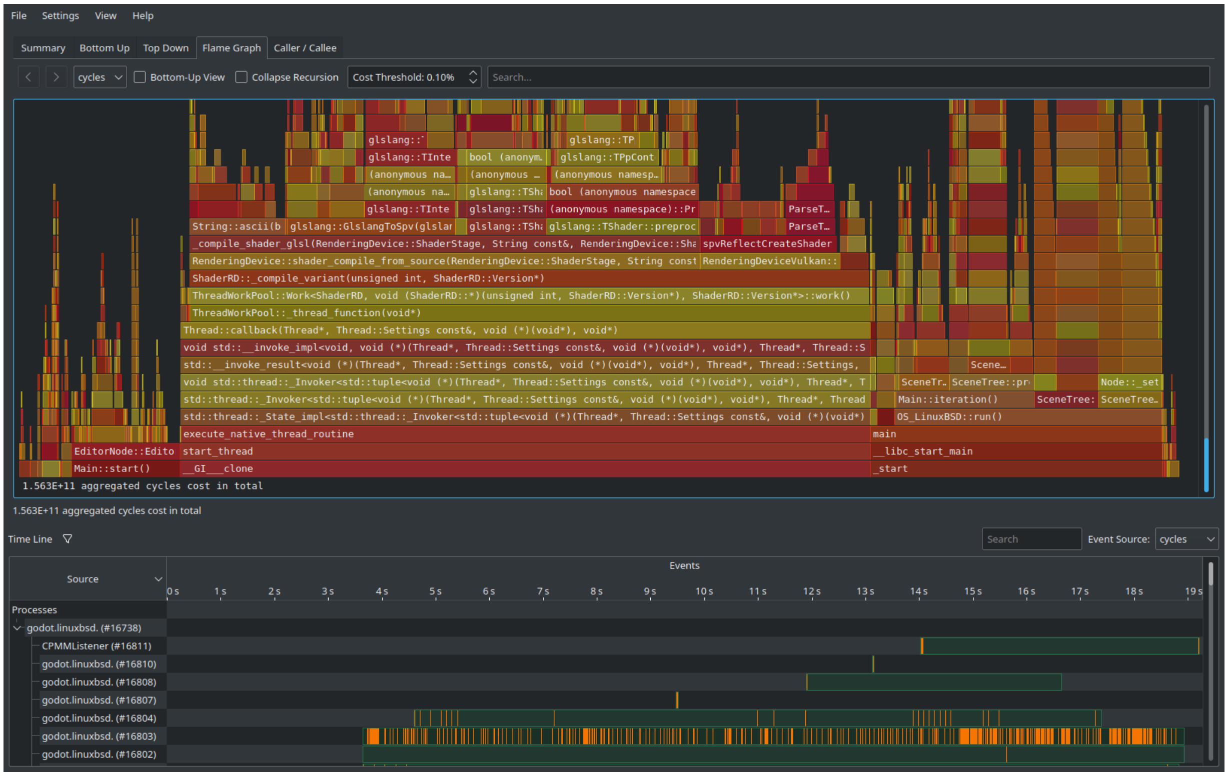
Event Type(s): cycles

Elevate Privileges: ☑

Off-CPU Profiling: ☐

```
querying current privileges...
kernel.kptr_restrict = 0
kernel.perf_event_paranoid = 2
/sys/kernel/debug 700
/sys/kernel/debug/tracing 700
```

- Use the tabs at the top to navigate between the different views. These views show the same data, but in different ways. The **Flame Graph** tab is a good way to see which functions take up the most time at a glance. These functions are therefore the most important ones to optimize, since optimizing them will improve performance the most.
- At the bottom of all tabs except **Summary**, you will also see a list of CPU threads started by the engine among with the CPU utilization for each thread. This lets you see threads that can be a bottleneck at a given point in time.


File
Settings
View
Help
Summary
Bottom Up
Top Down
Flame Graph
Caller / Callee
cycles
Bottom-Up View
Collapse Recursion
Cost Threshold: 0.10%
Search...
_compile_shader_glsl(RenderingDevice::ShaderStage, String const&, RenderingDevice::Sha
RenderingDevice::shader_compile_from_source(RenderingDevice::ShaderStage, String const
ShaderRD::_compile_variant(unsigned int, ShaderRD::Version*)
ThreadWorkPool::Work<ShaderRD, void (ShaderRD::*)(unsigned int, ShaderRD::Version*), ShaderRD::Version*>::work()
ThreadWorkPool::_thread_function(void*)
Thread::callback(Thread*, Thread::Settings const&, void (*)(void*), void*)
execute_native_thread_routine
start_thread
__GI___clone
main
__libc_start_main
_start
Main::iteration()
OS_LinuxBSD::run()
Main::start()
1.563E+11 aggregated cycles cost in total
Time Line
Search
Event Source:
cycles
Source
Events
Processes
godot.linuxbsd. (#16738)
CPMMListener (#16811)
godot.linuxbsd. (#16810)
godot.linuxbsd. (#16808)
godot.linuxbsd. (#16807)
godot.linuxbsd. (#16804)
godot.linuxbsd. (#16803)
godot.linuxbsd. (#16802)

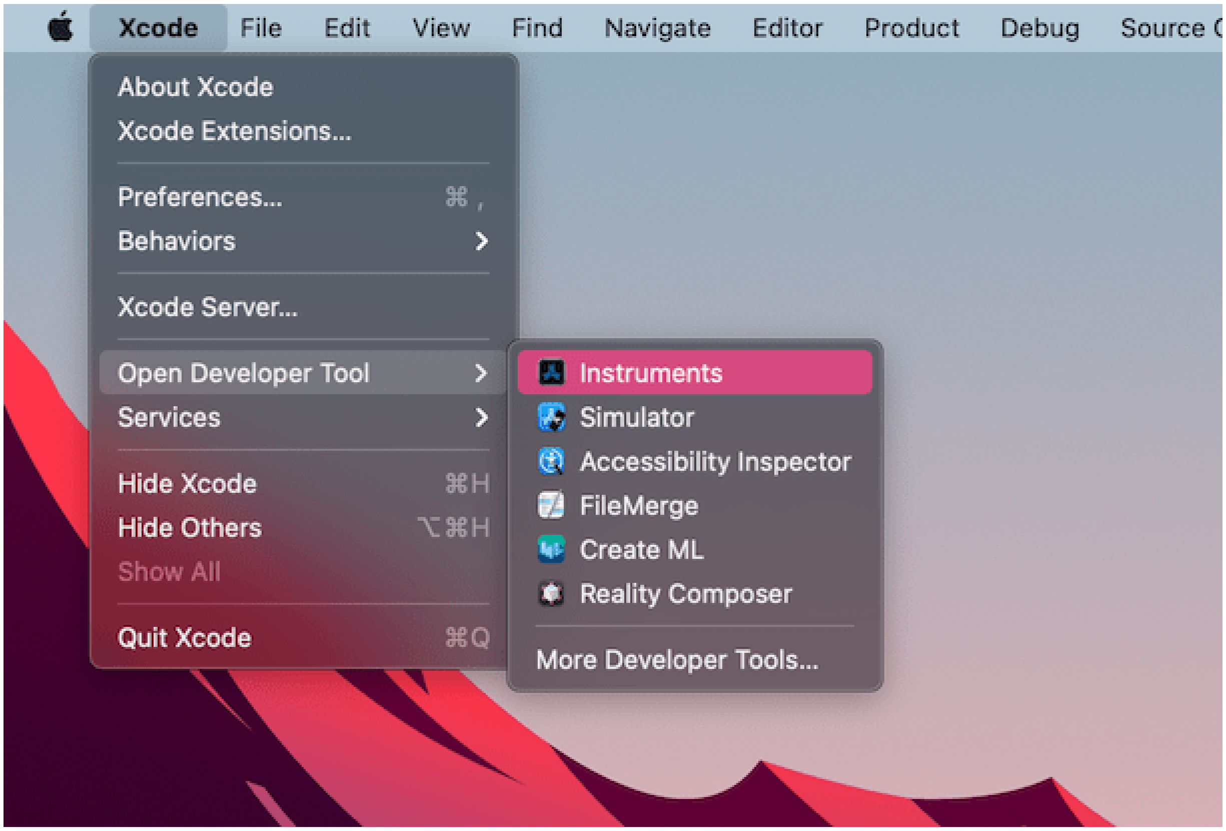
Note

If you don't want the startup procedure to be included in the profile, you can also attach HotSpot to a running process by clicking **Record Data** then setting the **Launch Application** dropdown option to **Attach To Process(es)**.

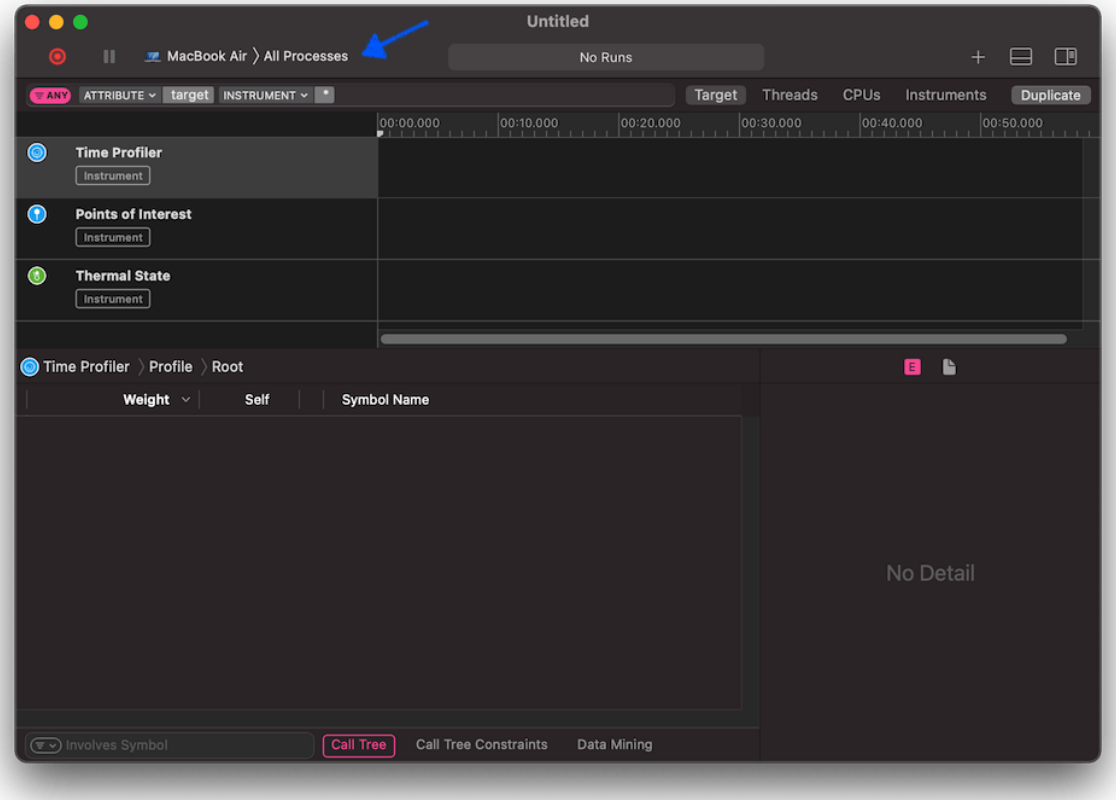
This process attachment-based workflow is similar to the one used by VerySleepy.

Xcode Instruments

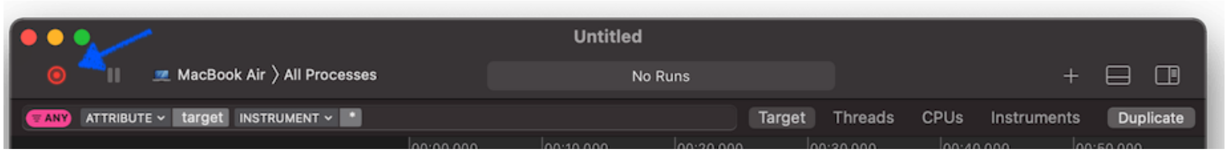
- Open Xcode. Select **Open Developer Tool** - **Instruments** from the **Xcode** app menu:
- Double-click on **Time Profiler** in the **Instruments** window:


Xcode
File
Edit
View
Find
Navigate
Editor
Product
Debug
Source C
About Xcode
Xcode Extensions...
Preferences... ⌘ ,
Behaviors
Xcode Server...
Open Developer Tool
Services
Hide Xcode ⌘H
Hide Others ⌥⌘H
Show All
Quit Xcode ⌘Q
Instruments
Simulator
Accessibility Inspector
FileMerge
Create ML
Reality Composer
More Developer Tools...

- In the Time Profiler window, click on the **Target** menu, select **Choose target...** and specify the path to the Godot binary, command line arguments and environment variables in the next window.

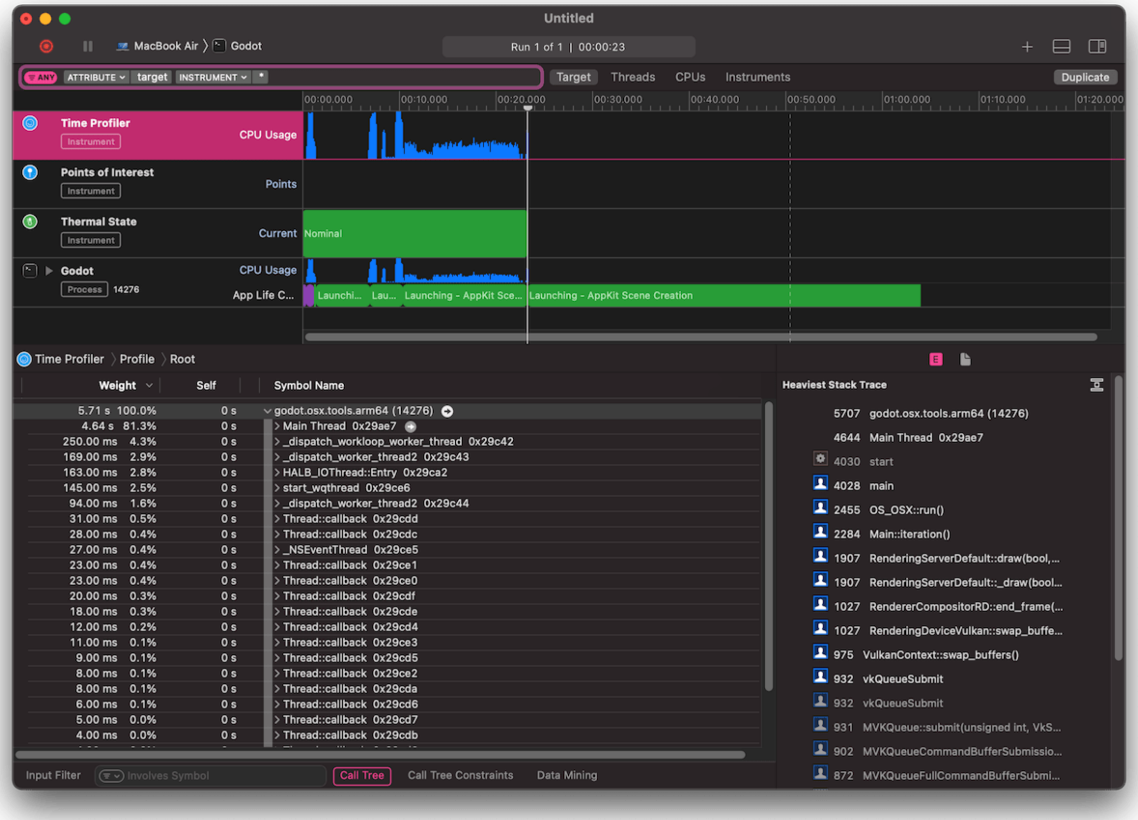

Untitled
MacBook Air 〉 All Processes
No Runs
ANY
ATTRIBUTE
target
INSTRUMENT
Target
Threads
CPUs
Instruments
Duplicate
00:00.000
00:10.000
00:20.000
00:30.000
00:40.000
00:50.000
Time Profiler
Instrument
Points of Interest
Instrument
Thermal State
Instrument
Time Profiler 〉 Profile 〉 Root
Weight
Self
Symbol Name
No Detail
Involves Symbol
Call Tree
Call Tree Constraints
Data Mining

- You can also attach the Time Profiler to a running process by selecting it from the **Target** menu.
- Click the **Start an immediate mode recording** button to start profiling.


Untitled
MacBook Air 〉 All Processes
No Runs
ANY
ATTRIBUTE
target
INSTRUMENT
Target
Threads
CPUs
Instruments
Duplicate

- Perform the actions you wish to profile in the editor or project. When you're done, click the **Stop** button.
- Wait for the results to appear.
- At the bottom of the window you will see a call tree for all CPU threads started, and the **Heaviest Stack Trace** overview.
- Select **Hide system libraries** in the **Call Tree** menu (at the bottom of window) to remove external modules.

- You can use the timeline at the top of the window to display details for the specific time period.


Untitled
MacBook Air 〉 Godot
Run 1 of 1 | 00:00:23
ANY
ATTRIBUTE
target
INSTRUMENT
Target
Threads
CPUs
Instruments
Duplicate
00:00.000
00:10.000
00:20.000
00:30.000
00:40.000
00:50.000
01:00.000
01:10.000
01:20.000
Time Profiler
Instrument
CPU Usage
Points of Interest
Instrument
Points
Thermal State
Instrument
Current
Nominal
Godot
Process
14276
CPU Usage
App Life C...
Launchi...
Lau...
Launching - AppKit Sce...
Launching - AppKit Scene Creation
Time Profiler 〉 Profile 〉 Root
Input Filter
Involves Symbol
Call Tree
Call Tree Constraints
Data Mining

Weight		Self	Symbol Name
5.71 s	100.0%	0 s	godot.osx.tools.arm64 (14276)
4.64 s	81.3%	0 s	Main Thread 0x29ae7
250.00 ms	4.3%	0 s	_dispatch_workloop_worker_thread 0x29c42
169.00 ms	2.9%	0 s	_dispatch_worker_thread2 0x29c43
163.00 ms	2.8%	0 s	HALB_IOThread::Entry 0x29ca2
145.00 ms	2.5%	0 s	start_wqthread 0x29ce6
94.00 ms	1.6%	0 s	_dispatch_worker_thread2 0x29c44
31.00 ms	0.5%	0 s	Thread::callback 0x29cdd
28.00 ms	0.4%	0 s	Thread::callback 0x29cdc
27.00 ms	0.4%	0 s	_NSEventThread 0x29ce5
23.00 ms	0.4%	0 s	Thread::callback 0x29ce1
23.00 ms	0.4%	0 s	Thread::callback 0x29ce0
20.00 ms	0.3%	0 s	Thread::callback 0x29cdf
18.00 ms	0.3%	0 s	Thread::callback 0x29cde
12.00 ms	0.2%	0 s	Thread::callback 0x29cd4
11.00 ms	0.1%	0 s	Thread::callback 0x29ce3
9.00 ms	0.1%	0 s	Thread::callback 0x29cd5
8.00 ms	0.1%	0 s	Thread::callback 0x29ce2
8.00 ms	0.1%	0 s	Thread::callback 0x29cda
6.00 ms	0.1%	0 s	Thread::callback 0x29cd6
5.00 ms	0.0%	0 s	Thread::callback 0x29cd7
4.00 ms	0.0%	0 s	Thread::callback 0x29cdb

Heaviest Stack Trace

5707	godot.osx.tools.arm64 (14276)
4644	Main Thread 0x29ae7
4030	start
4028	main
2455	OS_OSX::run()
2284	Main::iteration()
1907	RenderingServerDefault::draw(bool,...
1907	RenderingServerDefault::_draw(bool...
1027	RendererCompositorRD::end_frame(...
1027	RenderingDeviceVulkan::swap_buffe...
975	VulkanContext::swap_buffers()
932	vkQueueSubmit
932	vkQueueSubmit
931	MVKQueue::submit(unsigned int, VkS...
902	MVKQueueCommandBufferSubmissio...
872	MVKQueueFullCommandBufferSubmi...

Using sanitizers

What are sanitizers?

Sanitizers are static instrumentation tools that help find bugs that traditional debuggers usually cannot catch. This is particularly useful when combined with Unit testing in continuous integration.

Sanitizers can be used on Windows, macOS and Linux by using the Clang (LLVM), GCC or Visual Studio compilers. Certain platforms may also have their own sanitizers available. In situations where a single sanitizer is provided by several different compilers, remember that their output and behavior will differ slightly.

Using sanitizers on Godot

Sanitizers **require** recompiling the binary. This means you cannot use official Godot binaries to run sanitizers.

When compiling with any of the sanitizers enabled, the resulting binary will have the `.san` suffix added to its name to distinguish it from a binary without sanitizers.

There is a performance impact as many additional runtime checks need to be performed. Memory utilization will also increase. It is possible to enable certain combinations of multiple sanitizers in a single build. Beware of the performance impact when using multiple sanitizers at once though, as the resulting binary may be excessively slow.

Certain options can be passed to sanitizers without having to recompile the binary using environment variables.

Address sanitizer (ASAN)

- Available in Clang and GCC.
- **Supported platforms:** Linux, macOS, Windows (Visual Studio), Web
- Clang ASAN documentation [https://clang.llvm.org/docs/AddressSanitizer.html]

The address sanitizer is generally the most frequently used sanitizer. It can diagnose issues such as buffer overruns and out-of-bounds access. If the engine crashes with a message such as `free(): invalid pointer`, this is typically the result of a buffer overrun. (This message is printed by the C runtime, not Godot.)

In certain situations (such as detecting uninitialized memory reads), the address sanitizer doesn't suffice. The Memory sanitizer (MSAN) should be used instead.

It is also possible to detect use-after-return situations by specifying the `ASAN_OPTIONS=detect_stack_use_after_return=1` environment variable before *running* Godot (not when compiling it). This increases the address sanitizer's runtime overhead, so only enable this feature when you actually need it.

To enable the address sanitizer in a Godot build, pass the `use_asan=yes` SCons option when compiling. Enabling ASAN generally makes the resulting binary about 2× slower.

Warning

Due to a design decision [https://stackoverflow.com/questions/36971902/why-cant-clang-enable-all-sanitizers/], the address, memory and thread sanitizers are mutually exclusive. This means you can only use one of those sanitizers in a given binary.

Leak sanitizer (LSAN)

- Available in Clang and GCC.
- **Supported platforms:** Linux, Web
- Clang LSAN documentation [https://clang.llvm.org/docs/LeakSanitizer.html]

The leak sanitizer can detect memory leaks, which are situations where memory that is no longer in use is never freed by the running program. This can potentially lead to out-of-memory situations if the program runs for long enough. Since Godot may run on dedicated servers for months or even years without a restart, it's important to fix memory leaks when they occur.

To enable the leak sanitizer in a Godot build, pass the `use_lsan=yes` SCons option when compiling. Enabling LSAN only has a small performance overhead, but the program will be much slower to exit as leak detection occurs when the program exits.

Memory sanitizer (MSAN)

- Available in Clang only, not GCC.
- **Supported platforms:** Linux
- Clang MSAN documentation [https://clang.llvm.org/docs/MemorySanitizer.html]

The memory sanitizer complements the Address sanitizer (ASAN). Unlike the address sanitizer, the memory sanitizer can detect uninitialized memory reads.

To enable the memory sanitizer in a Godot build, pass the `use_msan=yes` SCons option when compiling. Enabling MSAN generally makes the resulting binary about 3× slower.

Warning

Due to a design decision [https://stackoverflow.com/questions/36971902/why-cant-clang-enable-all-sanitizers/], the address, memory and thread sanitizers are mutually exclusive. This means you can only use one of those sanitizers in a given binary.

Thread sanitizer (TSAN)

- Available in Clang and GCC.
- **Supported platforms:** Linux, macOS
- Clang TSAN documentation [https://clang.llvm.org/docs/ThreadSanitizer.html]

The thread sanitizer is used to track down race conditions related to multithreading. A race condition is when multiple threads try to modify the same data at the same time. Since thread scheduling can be ordered in any fashion by the operating system, this leads to incorrect behavior that only occurs occasionally (and can be difficult to track as a result). To prevent a race condition, you need to add a lock to ensure only one thread can access the shared data at a given time.

To enable the thread sanitizer in a Godot build, pass the `use_tsan=yes` SCons option when compiling. Enabling TSAN generally makes the resulting binary 10× slower, while also multiplying memory usage by an approximately 8× factor.

Warning

Due to a design decision [https://stackoverflow.com/questions/36971902/why-cant-clang-enable-all-sanitizers/], the address, memory and thread sanitizers are mutually exclusive. This means you can only use one of those sanitizers in a given binary.

Undefined behavior sanitizer (UBSAN)

- Available in Clang and GCC.
- **Supported platforms:** Linux, macOS, Web
- Clang UBSAN documentation [https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html]

The undefined behavior sanitizer is used to track down situations where the program exhibits random and unpredictable behavior. This is due to C/C++ code that is accepted by the compiler, but is not *correct*. Compiling with a different set of optimizations can also change the observed results of undefined behavior.

To enable the undefined behavior sanitizer in a Godot build, pass the `use_ubsan=yes` SCons option when compiling. Enabling UBSAN only has a small performance overhead.

Platform-specific sanitizers

Web

When compiling for the Web, there are 2 additional sanitizer SCons options available:

- `use_assertions=yes` enables runtime Emscripten assertions, which can catch various issues.
- `use_safe_heap=yes` enables Emscripten's SAFE_HEAP sanitizer [https://emscripten.org/docs/debugging/Sanitizers.html]. It provides similar functionality to ASAN, but it focuses on issues that are specific to WebAssembly. `SAFE_HEAP` is not guaranteed to be compatible with ASAN and UBSAN in the same binary, so you may have to build it separately.

Debugging on macOS

Debugging Godot editor

Attaching a debugger to the signed macOS process requires the "com.apple.security.get-task-allow" entitlement, which is not enabled by default, since apps can't be notarized as long as it is enabled. If you want to debug an official build of the editor it should be re-signed with the proper entitlements.

Create an `editor.entitlements` text file with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>com.apple.security.cs.allow-dyld-environment-variables</key>
        <true/>
        <key>com.apple.security.cs.allow-jit</key>
        <true/>
        <key>com.apple.security.cs.allow-unsigned-executable-memory</key>
        <true/>
        <key>com.apple.security.cs.disable-executable-page-protection</key>
        <true/>
        <key>com.apple.security.cs.disable-library-validation</key>
        <true/>
        <key>com.apple.security.device.audio-input</key>
        <true/>
        <key>com.apple.security.device.camera</key>
        <true/>
        <key>com.apple.security.get-task-allow</key>
        <true/>
    </dict>
</plist>
```

Then use the following command to re-sign the editor:

```
codesign -s - --deep --force --options=runtime --entitlements
./editor.entitlements ./path/to/Godot.app
```

Debugging exported project

To allow debugging, select the `codesign\debugging` (`com.apple.security.get-task-allow`) entitlement during the export. When it is selected, notarization is not supported and should be disabled.

Vulkan

- Validation layers

Validation layers

Validation layers enable developers to verify their application's correct use of the Vulkan API. Validation layers can be enabled in both debug and release builds, including in exported projects.

Note

Enabling validation layers has a performance impact, so only enable them when you actually need the output to debug the application.

Windows

Install the Vulkan SDK https://vulkan.lunarg.com/sdk/home, which contains validation layers as part of its default installation. No need to enable any optional features in the installer; installing the core Vulkan SDK suffices. You don't need to reboot after installing the SDK, but you may need to close and reopen your current terminal.

After installing the Vulkan SDK, run Godot with the `--gpu-validation` command line argument. You can also specify `--gpu-abort` which will make Godot quit as soon as a validation error happens. This can prevent your system from freezing if a validation error occurs.

macOS

Warning

Official Godot macOS builds do **not** support validation layers, as these are statically linked against the Vulkan SDK. Dynamic linking must be used instead.

In practice, this means that using validation layers on macOS **requires** you to use a Godot build compiled with the `use_volk=yes` SCons option. Compiling for macOS. If testing validation layers on an exported project, you must recompile the export template and specify it as a custom export template in your project's macOS export preset.

Install the Vulkan SDK https://vulkan.lunarg.com/sdk/home, which contains validation layers as part of its default installation. No need to enable any optional features in the installer; installing the core Vulkan SDK suffices. You don't need to reboot after installing the SDK, but you may need to close and reopen your current terminal.

After installing the Vulkan SDK, run a Godot binary that was compiled with `use_volk=yes` SCons option. Specify the `--gpu-validation` command line argument. You can also specify `--gpu-abort` which will make Godot quit as soon as a validation error happens. This can prevent your system from freezing if a validation error occurs.

Linux, *BSD

Install Vulkan validation layers from your distribution's repositories:

Alpine Linux

```
vulkan-validation-layers
```

Arch Linux

```
pacman -S vulkan-validation-layers
```

Debian/Ubuntu

```
apt install vulkan-validationlayers
```

Fedora

```
dnf install vulkan-validation-layers
```

FreeBSD

```
pkg install graphics/vulkan-validation-layers
```

Gentoo

```
emerge -an media-libs/vulkan-layers
```

Mageia

```
urpmi vulkan-validation-layers
```

OpenBSD

```
pkg_add graphics/vulkan-validation-layers
```

openSUSE

```
zypper install vulkan-validationlayers
```

Solus

```
eopkg install -c vulkan-validation-layers
```

You don't need to reboot after installing the validation layers, but you may need to close and reopen your current terminal.

After installing the package, run Godot with the `--gpu-validation` command line argument. You can also specify `--gpu-abort` which will make Godot quit as soon as a validation error happens. This can prevent your system from freezing if a validation error occurs.

Android

After enabling validation layers on Android, a developer can see errors and warning messages in the `adb logcat` output.

iOS

Validation layers are currently **not** supported on iOS.

Web

Validation layers are **not** supported on the web platform, as there is no support for Vulkan there.

Enabling validation layers

Build validation layers from official sources

To build Android libraries, follow the instructions on Khronos' repository [https://github.com/KhronosGroup/Vulkan-ValidationLayers/blob/master/BUILD.md#building-on-android]. After a successful build, the libraries will be located in `Vulkan-ValidationLayers/build-android/libs`.

Copy libraries

Copy libraries from `Vulkan-ValidationLayers/build-android/libs` to `godot/platform/android/java/app/libs/debug/vulkan_validation_layers`.

Your Godot source directory tree should look like on the example below:

```
godot
|-- platform
    |-- android
        |-- java
            |-- app
                |-- libs
                    |-- debug
                        |-- vulkan_validation_layers
                            |-- arm64-v8a
                            |-- armeabi-v7a
                            |-- x86
                            |-- x86_64
```

If the subdirectory `libs/debug/vulkan_validation_layers` doesn't exist, create it.

Compile and run the Android app

Linked validation layers are automatically loaded and enabled in Android debug builds. You can use Godot's One-click deploy feature to quickly test your project with the validation layers enabled.

Engine core and modules

The following pages are meant to introduce the global organization of Godot Engine's source code, and give useful tips for extending and fixing the engine on the C++ side.

Getting started with Godot's source code

This section covers the basics that you will encounter in (almost) every source file.

- Godot's architecture diagram
- Common engine methods and macros
- Core types
- Variant class
- Object class
- Inheritance class tree
- Internal rendering architecture
- 2D coordinate systems and 2D transforms
- Scripting development

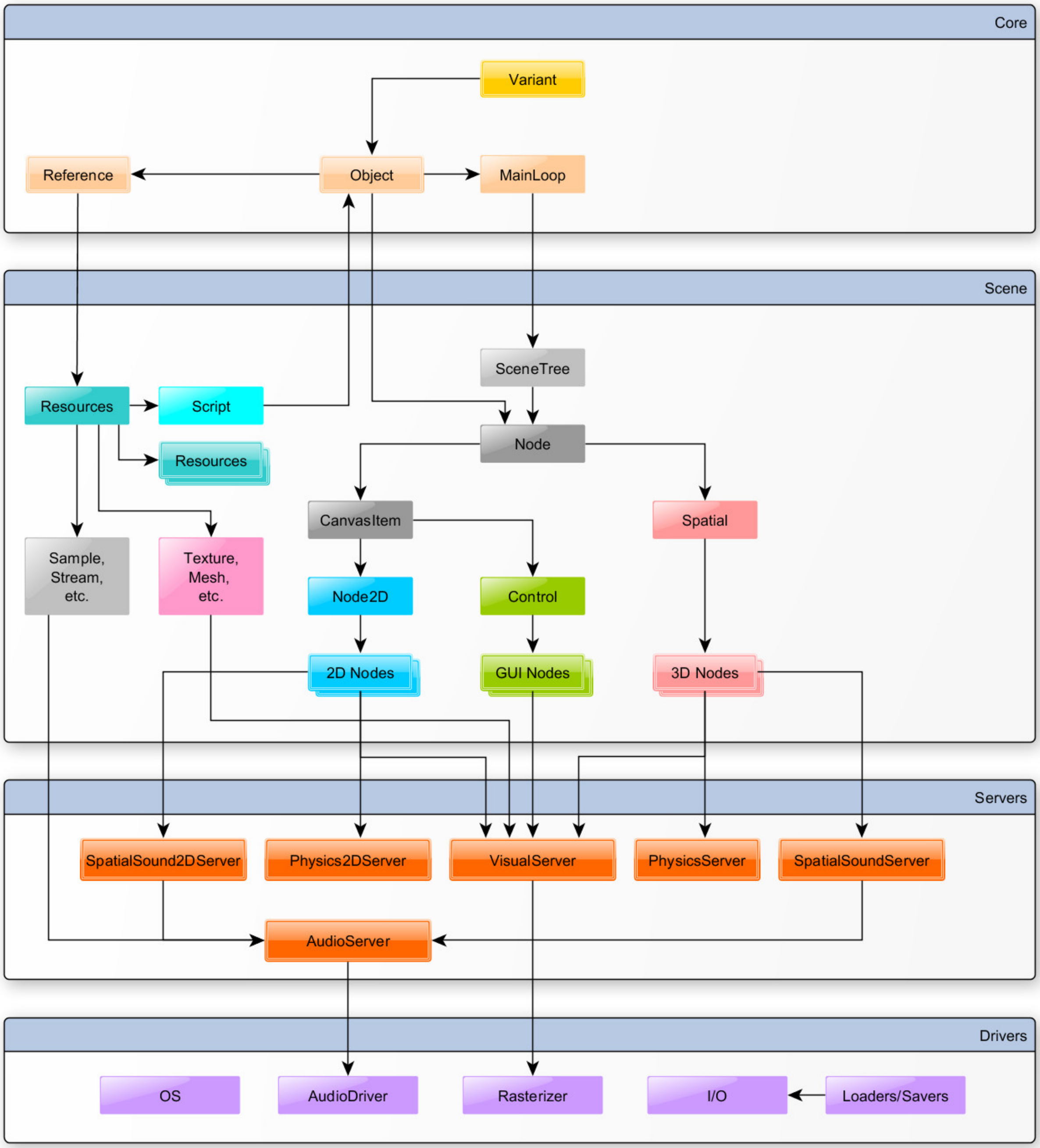
Extending Godot by modifying its source code

This section covers what you can do by modifying Godot's C++ source code.

- Custom modules in C++
- Binding to external libraries
- Custom Godot servers
- Custom resource format loaders
- Custom AudioStreams
- Custom platform ports
- Unit testing

Godot's architecture diagram

The following diagram describes the architecture used by Godot, from the core components down to the abstracted drivers, via the scene structure and the servers.


Core
Variant
Reference
Object
MainLoop
Scene
SceneTree
Resources
Script
Node
Resources
CanvasItem
Spatial
Sample,
Stream,
etc.
Texture,
Mesh,
etc.
Node2D
Control
2D Nodes
GUI Nodes
3D Nodes
Servers
SpatialSound2DServer
Physics2DServer
VisualServer
PhysicsServer
SpatialSoundServer
AudioServer
Drivers
OS
AudioDriver
Rasterizer
I/O
Loaders/Savers

Common engine methods and macros

Godot's C++ codebase makes use of dozens of custom methods and macros which are used in almost every file. This page is geared towards beginner contributors, but it can also be useful for those writing custom C++ modules.

Print text

```
// Prints a message to standard output.
print_line("Message");

// Non-String arguments are automatically converted to String
for printing.
// If passing several arguments, they will be concatenated
together with a
// space between each argument.
print_line("There are", 123, "nodes");

// Prints a message to standard output, but only when the
engine
// is started with the `--verbose` command line argument.
print_verbose("Message");

// Prints a rich-formatted message using BBCode to standard
output.
// This supports a subset of BBCode tags supported by
RichTextLabel
// and will also appear formatted in the editor Output panel.
// On Windows, this requires Windows 10 or later to work in the
terminal.
print_line_rich("[b]Bold[/b], [color=red]Red text[/color]")

// Prints a formatted error or warning message with a trace.
ERR_PRINT("Message");
```

```
WARN_PRINT("Message");

// Prints an error or warning message only once per session.
// This can be used to avoid spamming the console output.
ERR_PRINT_ONCE("Message");
WARN_PRINT_ONCE("Message");
```

If you need to add placeholders in your messages, use format strings as described below.

Format a string

The `vformat()` function returns a formatted String. It behaves in a way similar to C's `sprintf()`:

```
vformat("My name is %s.", "Godette");
vformat("%d bugs on the wall!", 1234);
vformat("Pi is approximately %f.", 3.1416);

// Converts the resulting String into a `const char *`.
// You may need to do this if passing the result as an argument
// to a method that expects a `const char *` instead of a
String.
vformat("My name is %s.", "Godette").c_str();
```

In most cases, try to use `vformat()` instead of string concatenation as it makes for more readable code.

Convert an integer or float to a string

This is not needed when printing numbers using `print_line()`, but you may still need to perform manual conversion for some other use cases.

```
// Stores the string "42" using integer-to-string conversion.
String int_to_string = itos(42);

// Stores the string "123.45" using real-to-string conversion.
String real_to_string = rtos(123.45);
```

Internationalize a string

There are two types of internationalization in Godot's codebase:

- `TTR()`: **Editor ("tools") translations** will only be processed in the editor. If a user uses the same text in one of their projects, it won't be translated if they provide a translation for it. When contributing to the engine, this is generally the macro you should use for localizable strings.
- `RTR()`: **Run-time translations** will be automatically localized in projects if they provide a translation for the given string. This kind of translation shouldn't be used in editor-only code.

```
// Returns the translated string that matches the user's locale settings.
// Translations are located in `editor/translations`.
// The localization template is generated automatically; don't modify it.
TTR("Exit the editor?");
```

To insert placeholders in localizable strings, wrap the localization macro in a `vformat()` call as follows:

```
String file_path = "example.txt";
vformat(TTR("Couldn't open \"%s\" for reading."), file_path);
```

Note

When using `vformat()` and a translation macro together, always wrap the translation macro in `vformat()`, not the other way around. Otherwise, the string will never match the translation as it will have the placeholder already replaced when it's passed to TranslationServer.

Clamp a value

Godot provides macros for clamping a value with a lower bound (`MAX`), an upper bound (`MIN`) or both (`CLAMP`):

```
int a = 3;
int b = 5;

MAX(b, 6); // 6
MIN(2, a); // 2
CLAMP(a, 10, 30); // 10
```

This works with any type that can be compared to other values (like `int` and `float`).

Microbenchmarking

If you want to benchmark a piece of code but don't know how to use a profiler, use this snippet:

```
uint64_t begin = Time::get_singleton()->get_ticks_usec();

// Your code here...

uint64_t end = Time::get_singleton()->get_ticks_usec();
print_line(vformat("Snippet took %d microseconds", end -
begin));
```

This will print the time spent between the `begin` declaration and the `end` declaration.

Note

You may have to `#include "core/os/os.h"` if it's not present already.

When opening a pull request, make sure to remove this snippet as well as the include if it wasn't there previously.

Get project/editor settings

There are four macros available for this:

```
// Returns the specified project setting's value,
// defaulting to `false` if it doesn't exist.
GLOBAL_DEF("section/subsection/value", false);

// Returns the specified editor setting's value,
// defaulting to "Untitled" if it doesn't exist.
EDITOR_DEF("section/subsection/value", "Untitled");
```

If a default value has been specified elsewhere, don't specify it again to avoid repetition:

```
// Returns the value of the project setting.
GLOBAL_GET("section/subsection/value");
// Returns the value of the editor setting.
EDITOR_GET("section/subsection/value");
```

It's recommended to use `GLOBAL_DEF`/`EDITOR_DEF` only once per setting and use `GLOBAL_GET`/`EDITOR_GET` in all other places where it's referenced.

Error macros

Godot features many error macros to make error reporting more convenient.

Warning

Conditions in error macros work in the **opposite** way of GDScript's built-in `assert()` function. An error is reached if the condition inside evaluates to `true`, not `false`.

Note

Only variants with custom messages are documented here, as these should always be used in new contributions. Make sure the custom message provided includes enough information for people to diagnose the issue, even if they don't know C++. In case a method was passed invalid arguments, you can print the invalid value in question to ease debugging.

For internal error checking where displaying a human-readable message isn't necessary, remove `_MSG` at the end of the macro name and don't supply a message argument.

Also, always try to return processable data so the engine can keep running well.

```
// Conditionally prints an error message and returns from the
function.
// Use this in methods which don't return a value.
ERR_FAIL_COND_MSG(!mesh.is_valid(), vformat("Couldn't load mesh
at: %s", path));

// Conditionally prints an error message and returns `0` from
the function.
// Use this in methods which must return a value.
ERR_FAIL_COND_V_MSG(rect.x < 0 || rect.y < 0, 0,
        "Couldn't calculate the rectangle's area.");

// Prints an error message if `index` is < 0 or >=
`SomeEnum::QUALITY_MAX`,
// then returns from the function.
```

```
ERR_FAIL_INDEX_MSG(index, SomeEnum::QUALITY_MAX,
        vformat("Invalid quality: %d. See SomeEnum for allowed values.", index));

// Prints an error message if `index` is < 0 >= `some_array.size()`,
// then returns `-1` from the function.
ERR_FAIL_INDEX_V_MSG(index, some_array.size(), -1,
        vformat("Item %d is out of bounds.", index));

// Unconditionally prints an error message and returns from the function.
// Only use this if you need to perform complex error checking.
if (!complex_error_checking_routine()) {
    ERR_FAIL_MSG("Couldn't reload the filesystem cache.");
}

// Unconditionally prints an error message and returns `false` from the function.
// Only use this if you need to perform complex error checking.
if (!complex_error_checking_routine()) {
    ERR_FAIL_V_MSG(false, "Couldn't parse the input arguments.");
}

// Crashes the engine. This should generally never be used
// except for testing crash handling code. Godot's philosophy
// is to never crash, both in the editor and in exported projects.
CRASH_NOW_MSG("Can't predict the future! Aborting.");
```

See also

See core/error/error_macros.h [https://github.com/godotengine/godot/blob/master/core/error/error_macros.h] in Godot's codebase for more information about each error macro.

Some functions return an error code (materialized by a return type of `Error`). This value can be returned directly from an error macro.

See the list of available error codes in core/error/error_list.h [https://github.com/godotengine/godot/blob/master/core/error/error_list.h].

Core types

Godot has a rich set of classes and templates that compose its core, and everything is built upon them.

This reference will try to list them in order for their better understanding.

Definitions

Godot uses the standard C99 datatypes, such as `uint8_t`, `uint32_t`, `int64_t`, etc. which are nowadays supported by every compiler. Reinventing the wheel for those is not fun, as it makes code more difficult to read.

In general, care is not taken to use the most efficient datatype for a given task unless using large structures or arrays. `int` is used through most of the code unless necessary. This is done because nowadays every device has at least a 32 bits bus and can do such operations in one cycle. It makes code more readable too.

For files or memory sizes, `size_t` is used, which is warranted to be 64 bits.

For Unicode characters, CharType instead of wchar_t is used, because many architectures have 4 bytes long wchar_t, where 2 bytes might be desired. However, by default, this has not been forced and CharType maps directly to wchar_t.

References:

- core/typedefs.h [https://github.com/godotengine/godot/blob/master/core/typedefs.h]

Memory model

PC is a wonderful architecture. Computers often have gigabytes of RAM, terabytes of storage and gigahertz of CPU, and when an application needs more resources the OS will swap out the inactive ones. Other architectures (like mobile or consoles) are in general more limited.

The most common memory model is the heap, where an application will request a region of memory, and the underlying OS will try to fit it somewhere and return it. This often works best and is flexible, but over time and with abuse, this can lead to segmentation.

Segmentation slowly creates holes that are too small for most common allocations, so that memory is wasted. There is a lot of literature about heap and segmentation, so this topic will not be developed further here. Modern operating systems use paged memory, which helps mitigate the problem of segmentation but doesn't solve it.

However, in many studies and tests, it is shown that given enough memory, if the maximum allocation size is below a given threshold in proportion to the maximum heap size and proportion of memory intended to be unused, segmentation will not be a problem over time as it will remain constant. In other words, leave 10-20% of your memory free and perform all small allocations and you are fine.

Godot ensures that all objects that can be allocated dynamically are small (less than a few kb at most). But what happens if an allocation is too large (like an image or mesh geometry or large array)? In this case Godot has the option to use a dynamic memory pool. This memory needs to be locked to be accessed, and if an allocation runs out of memory, the pool will be rearranged and compacted on demand. Depending on the need of the game, the programmer can configure the dynamic memory pool size.

Allocating memory

Godot has many tools for tracking memory usage in a game, especially during debug. Because of this, the regular C and C++ library calls should not be used. Instead, a few other ones are provided.

For C-style allocation, Godot provides a few macros:

```
memalloc()
memrealloc()
memfree()
```

These are equivalent to the usual malloc, realloc, free of the standard C library.

For C++-style allocation, special macros are provided:

```
memnew( Class / Class(args) )
memdelete( instance )

memnew_arr( Class , amount )
memdelete_arr( pointer to array )
```

which are equivalent to new, delete, new[] and delete[].

memnew/memdelete also use a little C++ magic and notify Objects right after they are created, and right before they are deleted.

For dynamic memory, the PoolVector<> template is provided. PoolVector is a standard vector class, and is very similar to vector in the C++ standard library. To create a PoolVector buffer, use this:

```
PoolVector<int> data;
```

PoolVector can be accessed using the [] operator and a few helpers exist for this:

```
PoolVector<int>::Read r = data.read()
int someint = r[4]
```

```
PoolVector<int>::Write w = data.write()
w[4] = 22;
```

These operations allow fast read/write from PoolVectors and keep it locked until they go out of scope. However, PoolVectors should be used for small, dynamic memory operations, as read() and write() are too slow for a large amount of accesses.

References:

- core/os/memory.h [https://github.com/godotengine/godot/blob/master/core/os/memory.h]
- core/pool_vector.h [https://github.com/godotengine/godot/blob/master/core/pool_vector.cpp]

Containers

Godot provides also a set of common containers:

- Vector
- List
- Set
- Map

They aim to be as minimal as possible, as templates in C++ are often inlined and make the binary size much fatter, both in debug symbols and code. List, Set and Map can be iterated using pointers, like this:

```
for(List<int>::Element *E=somelist.front();E;E=E->next()) {
    print_line(E->get()); // print the element
}
```

The Vector<> class also has a few nice features:

- It does copy on write, so making copies of it is cheap as long as they are not modified.
- It supports multi-threading, by using atomic operations on the reference counter.

References:

- core/templates/vector.h [https://github.com/godotengine/godot/blob/master/core/templates/vector.h]
- core/templates/list.h [https://github.com/godotengine/godot/blob/master/core/templates/list.h]
- core/templates/set.h [https://github.com/godotengine/godot/blob/master/core/templates/hash_set.h]
- core/templates/map.h [https://github.com/godotengine/godot/blob/master/core/templates/hash_map.h]

String

Godot also provides a String class. This class has a huge amount of features, full Unicode support in all the functions (like case operations) and utf8 parsing/extracting, as well as helpers for conversion and visualization.

References:

- core/string/ustring.h [https://github.com/godotengine/godot/blob/master/core/string/ustring.h]

StringName

StringNames are like a String, but they are unique. Creating a StringName from a string results in a unique internal pointer for all equal strings. StringNames are useful for using strings as identifier, as comparing them is basically comparing a pointer.

Creation of a StringName (especially a new one) is slow, but comparison is fast.

References:

- core/string/string_name.h [https://github.com/godotengine/godot/blob/master/core/string/string_name.h]

Math types

There are several linear math types available in the core/math directory.

References:

- core/math [https://github.com/godotengine/godot/tree/master/core/math]

NodePath

This is a special datatype used for storing paths in a scene tree and referencing them fast.

References:

- core/string/node_path.h [https://github.com/godotengine/godot/blob/master/core/string/node_path.h]

RID

RIDs are resource IDs. Servers use these to reference data stored in them. RIDs are opaque, meaning that the data they reference can't be accessed directly. RIDs are unique, even for different types of referenced data.

References:

- core/templates/rid.h [https://github.com/godotengine/godot/blob/master/core/templates/rid.h]

Variant class

About

Variant is the most important datatype of Godot, it's the most important class in the engine. A Variant takes up only 20 bytes and can store almost any engine datatype inside of it. Variants are rarely used to hold information for long periods of time, instead they are used mainly for communication, editing, serialization and generally moving data around.

A Variant can:

- Store almost any datatype
- Perform operations between many variants (GDScript uses Variant as its atomic/native datatype).
- Be hashed, so it can be compared quickly to other variants
- Be used to convert safely between datatypes
- Be used to abstract calling methods and their arguments (Godot exports all its functions through variants)
- Be used to defer calls or move data between threads.
- Be serialized as binary and stored to disk, or transferred via network.
- Be serialized to text and use it for printing values and editable settings.
- Work as an exported property, so the editor can edit it universally.
- Be used for dictionaries, arrays, parsers, etc.

Basically, thanks to the Variant class, writing Godot itself was a much, much easier task, as it allows for highly dynamic constructs not common of C++ with little effort. Become a friend of Variant today.

References:

- core/variant/variant.h [https://github.com/godotengine/godot/blob/master/core/variant/variant.h]

Containers: Dictionary and Array

Both are implemented using variants. A Dictionary can match any datatype used as key to any other datatype. An Array just holds an array of Variants. Of course, a Variant can also hold a Dictionary and an Array inside, making it even more flexible.

Modifications to a container will modify all references to it. A Mutex should be created to lock it if multi threaded access is desired.

Copy-on-write (COW) mode support for containers was dropped with Godot 3.0.

References:

- core/variant/dictionary.h [https://github.com/godotengine/godot/blob/master/core/variant/dictionary.h]
- core/variant/array.h [https://github.com/godotengine/godot/blob/master/core/variant/array.h]

Object class

See also

This page describes the C++ implementation of objects in Godot. Looking for the Object class reference? Have a look here.

General definition

Object is the base class for almost everything. Most classes in Godot inherit directly or indirectly from it. Objects provide reflection and editable properties, and declaring them is a matter of using a single macro like this:

```
class CustomObject : public Object {

    GDCLASS(CustomObject, Object); // this is required to inherit
};
```

This adds a lot of functionality to Objects. For example:

```
obj = memnew(CustomObject);
print_line("Object class: ", obj->get_class()); // print object class

obj2 = Object::cast_to<OtherClass>(obj); // converting between classes, this also works without RTTI enabled.
```

References:

- core/object/object.h [https://github.com/godotengine/godot/blob/master/core/object/object.h]

Registering an Object

ClassDB is a static class that holds the entire list of registered classes that inherit from Object, as well as dynamic bindings to all their methods properties and integer constants.

Classes are registered by calling:

```
ClassDB::register_class<MyCustomClass>()
```

Registering it will allow the class to be instanced by scripts, code, or creating them again when deserializing.

Registering as virtual is the same but it can't be instanced.

```
ClassDB::register_virtual_class<MyCustomClass>()
```

Object-derived classes can override the static function `static void _bind_methods()`. When one class is registered, this static function is called to register all the object methods, properties, constants, etc. It's only called once. If an Object derived class is instanced but has not been registered, it will be registered as virtual automatically.

Inside `_bind_methods`, there are a couple of things that can be done. Registering functions is one:

```
ClassDB::bind_method(D_METHOD("methodname", "arg1name",
"arg2name", "arg3name"), &MyCustomType::method);
```

Default values for arguments can be passed as parameters at the end:

```
ClassDB::bind_method(D_METHOD("methodname", "arg1name",
"arg2name", "arg3name"), &MyCustomType::method, DEFVAL(-1),
DEFVAL(-2)); // Default values for arg2name (-1) and arg3name
(-2).
```

Default values must be provided in the same order as they are declared, skipping required arguments and then providing default values for the optional ones. This matches the syntax for declaring methods in C++.

`D_METHOD` is a macro that converts "methodname" to a StringName for more efficiency. Argument names are used for introspection, but when compiling on release, the macro ignores them, so the strings are unused and optimized away.

Check `_bind_methods` of Control or Object for more examples.

If just adding modules and functionality that is not expected to be documented as thoroughly, the `D_METHOD()` macro can safely be ignored and a string passing the name can be passed for brevity.

References:

- core/object/class_db.h [https://github.com/godotengine/godot/blob/master/core/object/class_db.h]

Constants

Classes often have enums such as:

```
enum SomeMode {
   MODE_FIRST,
   MODE_SECOND
};
```

For these to work when binding to methods, the enum must be declared convertible to int. A macro is provided to help with this:

```
VARIANT_ENUM_CAST(MyClass::SomeMode); // now functions that
take SomeMode can be bound.
```

The constants can also be bound inside `_bind_methods`, by using:

```
BIND_CONSTANT(MODE_FIRST);
BIND_CONSTANT(MODE_SECOND);
```

Properties (set/get)

Objects export properties, properties are useful for the following:

- Serializing and deserializing the object.
- Creating a list of editable values for the Object derived class.

Properties are usually defined by the PropertyInfo() class and constructed as:

```
PropertyInfo(type, name, hint, hint_string, usage_flags)
```

For example:

```
PropertyInfo(Variant::INT, "amount", PROPERTY_HINT_RANGE,
"0,49,1", PROPERTY_USAGE_EDITOR)
```

This is an integer property named "amount". The hint is a range, and the range goes from 0 to 49 in steps of 1 (integers). It is only usable for the editor (editing the value visually) but won't be serialized.

Another example:

```
PropertyInfo(Variant::STRING, "modes", PROPERTY_HINT_ENUM,
"Enabled,Disabled,Turbo")
```

This is a string property, can take any string but the editor will only allow the defined hint ones. Since no usage flags were specified, the default ones are PROPERTY_USAGE_STORAGE and PROPERTY_USAGE_EDITOR.

There are plenty of hints and usage flags available in object.h, give them a check.

Properties can also work like C# properties and be accessed from script using indexing, but this usage is generally discouraged, as using functions is preferred for legibility. Many properties are also bound with categories, such as "animation/frame" which also make indexing impossible unless using operator [].

From `_bind_methods()`, properties can be created and bound as long as set/get functions exist. Example:

```
ADD_PROPERTY(PropertyInfo(Variant::INT, "amount"),
"set_amount", "get_amount")
```

This creates the property using the setter and the getter.

Binding properties using `_set`/`_get`/`_get_property_list`

An additional method of creating properties exists when more flexibility is desired (i.e. adding or removing properties on context).

The following functions can be overridden in an Object derived class, they are NOT virtual, DO NOT make them virtual, they are called for every override and the previous ones are not invalidated (multilevel call).

```
protected:
     void _get_property_list(List<PropertyInfo> *r_props)
const;       // return list of properties
     bool _get(const StringName &p_property, Variant &r_value)
const; // return true if property was found
     bool _set(const StringName &p_property, const Variant
&p_value); // return true if property was found
```

This is also a little less efficient since `p_property` must be compared against the desired names in serial order.

Dynamic casting

Godot provides dynamic casting between Object-derived classes, for example:

```
void somefunc(Object *some_obj) {

     Button *button = Object::cast_to<Button>(some_obj);
}
```

If cast fails, NULL is returned. This system uses RTTI, but it also works fine (although a bit slower) when RTTI is disabled. This is useful on platforms where a small binary size is ideal, such as HTML5 or consoles (with low memory footprint).

Signals

Objects can have a set of signals defined (similar to Delegates in other languages). This example shows how to connect to them:

```
obj->connect(<signal>, target_instance, target_method)
// for example:
obj->connect("enter_tree", this, "_node_entered_tree")
```

The method `_node_entered_tree` must be registered to the class using `ClassDB::bind_method` (explained before).

Adding signals to a class is done in `_bind_methods`, using the `ADD_SIGNAL` macro, for example:

```
ADD_SIGNAL(MethodInfo("been_killed"))
```

Notifications

All objects in Godot have a _notification method that allows it to respond to engine level callbacks that may relate to it. More information can be found on the Godot notifications page.

References

RefCounted inherits from Object and holds a reference count. It is the base for reference counted object types. Declaring them must be done using Ref<> template. For example:

```
class MyReference: public RefCounted {
    GDCLASS(MyReference, RefCounted);
};

Ref<MyReference> myref(memnew(MyReference));
```

`myref` is reference counted. It will be freed when no more Ref<> templates point to it.

References:

- core/object/reference.h [https://github.com/godotengine/godot/blob/master/core/object/ref_counted.h]

Resources

Resource inherits from Reference, so all resources are reference counted. Resources can optionally contain a path, which reference a file on disk. This can be set with `resource.set_path(path)`, though this is normally done by the resource loader. No two different resources can have the same path; attempting to do so will result in an error.

Resources without a path are fine too.

References:

- core/io/resource.h [https://github.com/godotengine/godot/blob/master/core/io/resource.h]

Resource loading

Resources can be loaded with the ResourceLoader API, like this:

```
Ref<Resource> res =
ResourceLoader::load("res://someresource.res")
```

If a reference to that resource has been loaded previously and is in memory, the resource loader will return that reference. This means that there can be only one resource loaded from a file referenced on disk at the same time.

- resourceinteractiveloader (TODO)

References:

- core/io/resource_loader.h [https://github.com/godotengine/godot/blob/master/core/io/resource_loader.h]

Resource saving

Saving a resource can be done with the resource saver API:

```
ResourceSaver::save("res://someresource.res", instance)
```

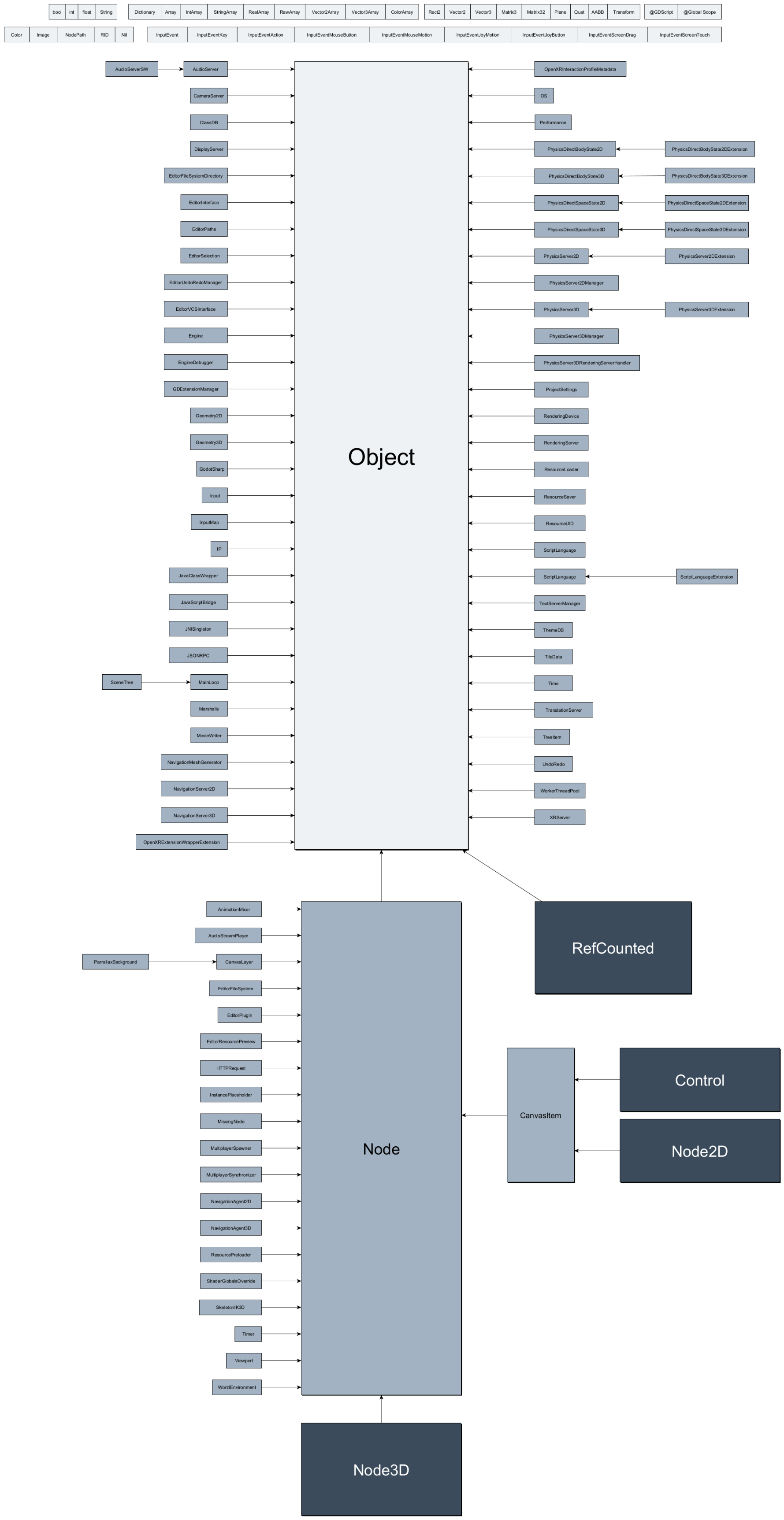
The instance will be saved, and sub resources that have a path to a file will be saved as a reference to that resource. Sub resources without a path will be bundled with the saved resource and assigned sub-IDs, like `res://someresource.res::1`. This also helps to cache them when loaded.

References:

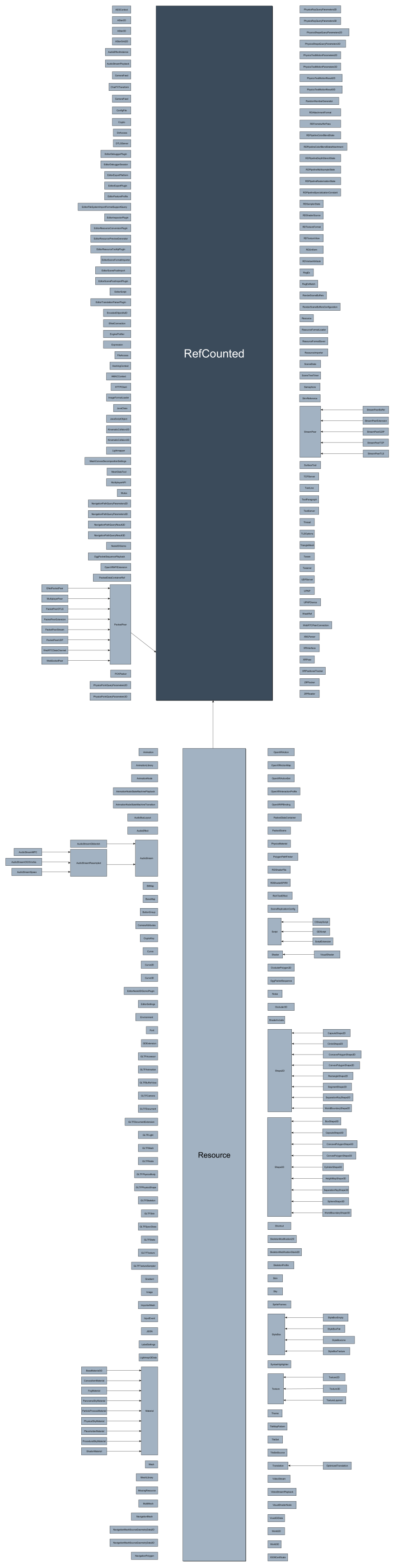
- core/io/resource_saver.h [https://github.com/godotengine/godot/blob/master/core/io/resource_saver.h]

Inheritance class tree

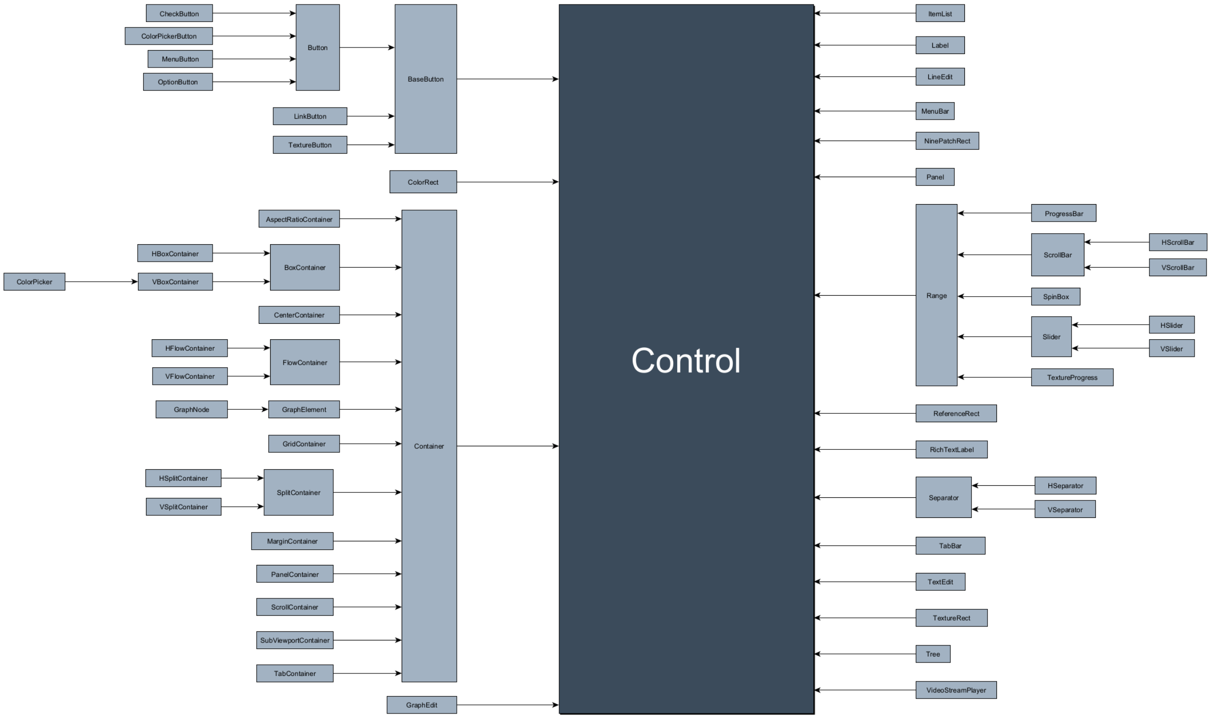
Object


bool
int
float
String
Dictionary
Array
IntArray
StringArray
RealArray
RawArray
Vector2Array
Vector3Array
ColorArray
Rect2
Vector2
Vector3
Matrix3
Matrix32
Plane
Quat
AABB
Transform
@GDScript
@Global Scope
Color
Image
NodePath
RID
Nil
InputEvent
InputEventKey
InputEventAction
InputEventMouseButton
InputEventMouseMotion
InputEventJoyMotion
InputEventJoyButton
InputEventScreenDrag
InputEventScreenTouch
AudioServerSW
AudioServer
CameraServer
ClassDB
DisplayServer
EditorFileSystemDirectory
EditorInterface
EditorPaths
EditorSelection
EditorUndoRedoManager
EditorVCSInterface
Engine
EngineDebugger
GDExtensionManager
Geometry2D
Geometry3D
GodotSharp
Input
InputMap
IP
JavaClassWrapper
JavaScriptBridge
JNISingleton
JSONRPC
SceneTree
MainLoop
Marshalls
MovieWriter
NavigationMeshGenerator
NavigationServer2D
NavigationServer3D
OpenXRExtensionWrapperExtension
Object
OpenXRInteractionProfileMetadata
OS
Performance
PhysicsDirectBodyState2D
PhysicsDirectBodyState2DExtension
PhysicsDirectBodyState3D
PhysicsDirectBodyState3DExtension
PhysicsDirectSpaceState2D
PhysicsDirectSpaceState2DExtension
PhysicsDirectSpaceState3D
PhysicsDirectSpaceState3DExtension
PhysicsServer2D
PhysicsServer2DExtension
PhysicsServer2DManager
PhysicsServer3D
PhysicsServer3DExtension
PhysicsServer3DManager
PhysicsServer3DRenderingServerHandler
ProjectSettings
RenderingDevice
RenderingServer
ResourceLoader
ResourceSaver
ResourceUID
ScriptLanguage
ScriptLanguage
ScriptLanguageExtension
TextServerManager
ThemeDB
TileData
Time
TranslationServer
TreeItem
UndoRedo
WorkerThreadPool
XRServer
AnimationMixer
AudioStreamPlayer
ParrallaxBackground
CanvasLayer
EditorFileSystem
EditorPlugin
EditorResourcePreview
HTTPRequest
InstancePlaceholder
MissingNode
MultiplayerSpawner
MultiplayerSynchronizer
NavigationAgent2D
NavigationAgent3D
ResourcePreloader
ShaderGlobalsOverride
SkeletonIK3D
Timer
Viewport
WorldEnvironment
Node
RefCounted
CanvasItem
Control
Node2D
Node3D

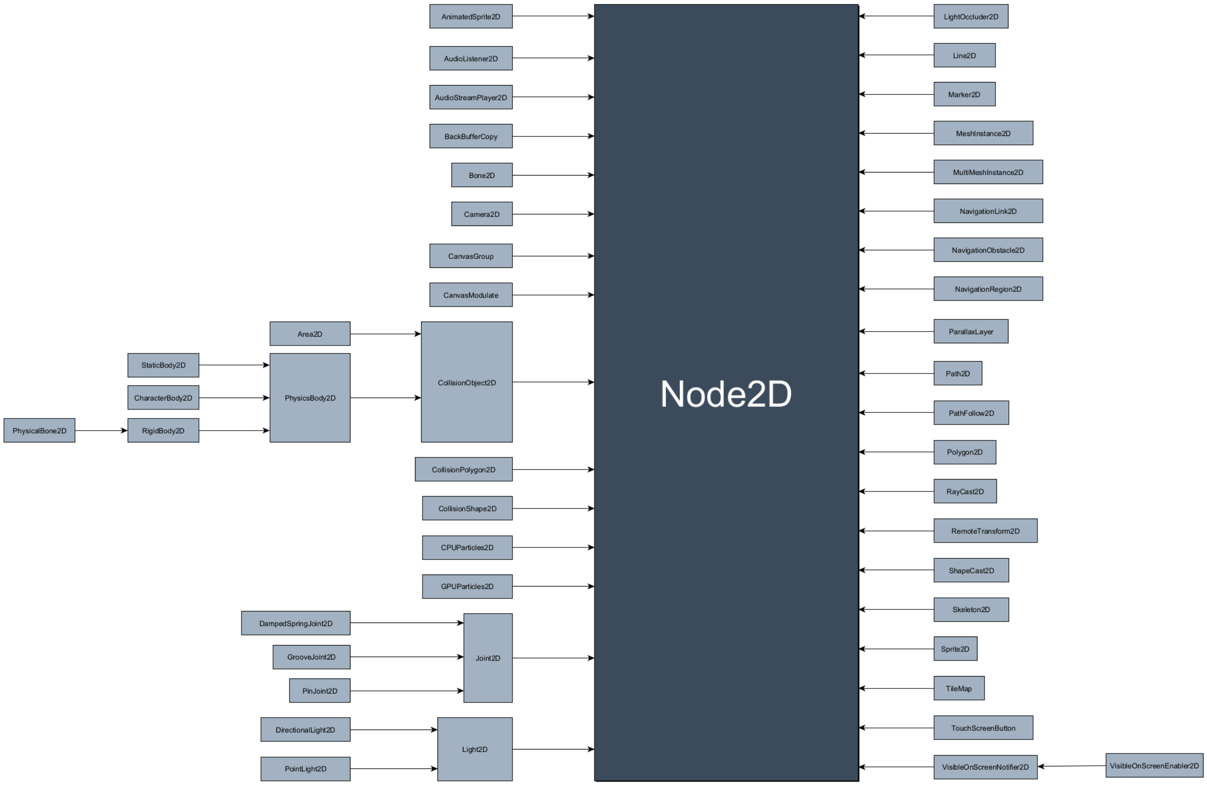
Reference


RefCounted
AESContext
AStar2D
AStar3D
AStarGrid2D
AudioEffectInstance
AudioStreamPlayback
CameraFeed
CharFXTransform
CameraFeed
ConfigFile
Crypto
DirAccess
DTLSServer
EditorDebuggerPlugin
EditorDebuggerSession
EditorExportPlatform
EditorExportPlugin
EditorFeatureProfile
EditorFileSystemImportFormatSupportQuery
EditorInspectorPlugin
EditorResourceConversionPlugin
EditorResourcePreviewGenerator
EditorResourceTooltipPlugin
EditorSceneFormatImporter
EditorScenePostImport
EditorScenePostImportPlugin
EditorScript
EditorTranslationParserPlugin
EncodedObjectAsID
ENetConnection
EngineProfiler
Expression
FileAccess
HashingContext
HMACContext
HTTPClient
ImageFormatLoader
JavaClass
JavaScriptObject
KinematicCollision2D
KinematicCollision3D
Lightmapper
MeshConvexDecompositionSettings
MeshDataTool
MultiplayerAPI
Mutex
NavigationPathQueryParameters2D
NavigationPathQueryParameters3D
NavigationPathQueryResult2D
NavigationPathQueryResult3D
Node3DGizmo
OggPacketSequencePlayback
OpenXRAPIExtension
PackedDataContainerRef
ENetPacketPeer
MultiplayerPeer
PacketPeerDTLS
PacketPeerExtension
PacketPeerStream
PacketPeerUDP
WebRTCDataChannel
WebSocketPeer
PacketPeer
PCKPacker
PhysicsPointQueryParameters2D
PhysicsPointQueryParameters3D
PhysicsRayQueryParameters2D
PhysicsRayQueryParameters3D
PhysicsShapeQueryParameters2D
PhysicsShapeQueryParameters3D
PhysicsTestMotionParameters2D
PhysicsTestMotionParameters3D
PhysicsTestMotionResult2D
PhysicsTestMotionResult3D
RandomNumberGenerator
RDAttachmentFormat
RDFramebufferPass
RDPipelineColorBlendState
RDPipelineColorBlendStateAttachment
RDPipelineDepthStencilState
RDPipelineMultisampleState
RDPipelineRasterizationState
RDPipelineSpecializationConstant
RDSamplerState
RDShaderSource
RDTextureFormat
RDTextureView
RDUniform
RDVertexAttribute
RegEx
RegExMatch
RenderSceneBuffers
RenderSceneBuffersConfiguration
Resource
ResourceFormatLoader
ResourceFormatSaver
ResourceImporter
SceneState
SceneTreeTimer
Semaphore
SkinReference
StreamPeer
StreamPeerBuffer
StreamPeerExtension
StreamPeerGZIP
StreamPeerTCP
StreamPeerTLS
SurfaceTool
TCPServer
TextLine
TextParagraph
TextServer
Thread
TLSOptions
TriangleMesh
Tween
Tweener
UDPServer
UPNP
UPNPDevice
WeakRef
WebRTCPeerConnection
XMLParser
XRInterface
XRPose
XRPositionalTracker
ZIPPacker
ZIPReader
Resource
Animation
AnimationLibrary
AnimationNode
AnimationNodeStateMachinePlayback
AnimationNodeStateMachineTransition
AudioBusLayout
AudioEffect
AudioStreamGibberish
AudioStreamMPC
AudioStreamOGGVorbis
AudioStreamSpeex
AudioStreamResampled
AudioStream
BitMap
BoneMap
ButtonGroup
CameraAttributes
CryptoKey
Curve
Curve2D
Curve3D
EditorNode3DGizmoPlugin
EditorSettings
Environment
Font
GDExtension
GLTFAccessor
GLTFAnimation
GLTFBufferView
GLTFCamera
GLTFDocument
GLTFDocumentExtension
GLTFLight
GLTFMesh
GLTFNode
GLTFPhysicsBody
GLTFPhysicsShape
GLTFSkeleton
GLTFSkin
GLTFSpecGloss
GLTFState
GLTFTexture
GLTFTextureSampler
Gradient
Image
ImporterMesh
InputEvent
JSON
LabelSettings
LightmapGIData
BaseMaterial3D
CanvasItemMaterial
FogMaterial
PanoramaSkyMaterial
ParticleProcessMaterial
PhysicalSkyMaterial
PlaceholderMaterial
ProceduralSkyMaterial
ShaderMaterial
Material
Mesh
MeshLibrary
MissingResource
MultiMesh
NavigationMesh
NavigationMeshSourceGeometryData2D
NavigationMeshSourceGeometryData3D
NavigationPolygon
OpenXRAction
OpenXRActionMap
OpenXRActionSet
OpenXRInteractionProfile
OpenXRIPBinding
PackedDataContainer
PackedScene
PhysicsMaterial
PolygonPathFinder
RDShaderFile
RDShaderSPIRV
RichTextEffect
SceneReplicationConfig
Script
CSharpScript
GDScript
ScriptExtension
Shader
VisualShader
OccluderPolygon2D
OggPacketSequence
Noise
Occluder3D
ShaderInclude
Shape2D
CapsuleShape2D
CircleShape2D
ConcavePolygonShape2D
ConvexPolygonShape2D
RectangleShape2D
SegmentShape2D
SeparationRayShape2D
WorldBoundaryShape2D
Shape3D
BoxShape3D
CapsuleShape3D
ConcavePolygonShape3D
ConvexPolygonShape3D
CylinderShape3D
HeightMapShape3D
SeparationRayShape3D
SphereShape3D
WorldBoundaryShape3D
Shortcut
SkeletonModification2D
SkeletonModificationStack2D
SkeletonProfile
Skin
Sky
SpriteFrames
StyleBox
StyleBoxEmpty
StyleBoxFlat
StyleBoxLine
StyleBoxTexture
SyntaxHighlighter
Texture
Texture2D
Texture3D
TextureLayered
Theme
TileMapPattern
TileSet
TileSetSource
Translation
OptimizedTranslation
VideoStream
VideoStreamPlayback
VisualShaderNode
VoxelGIData
World2D
World3D
X509Certificate

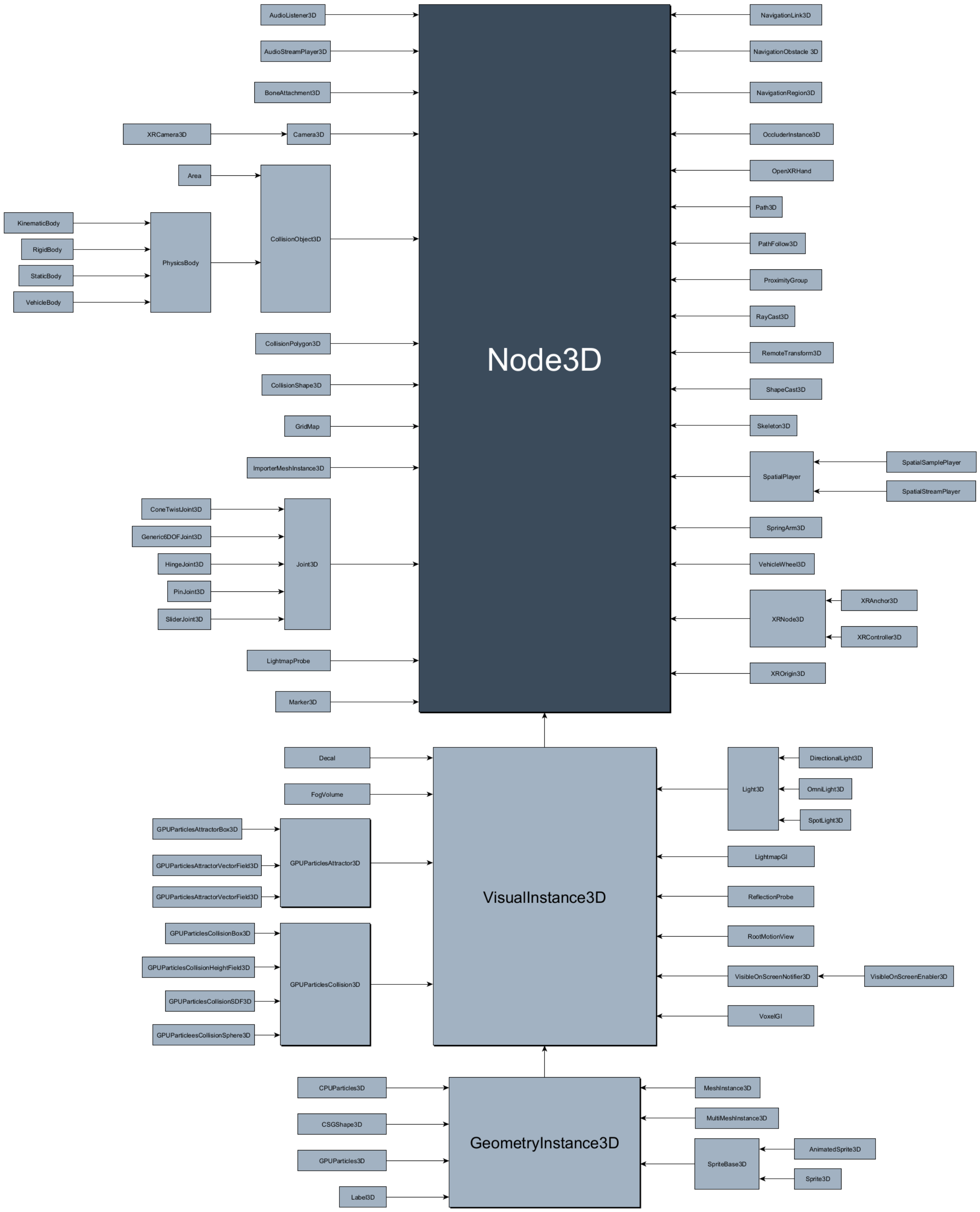
Control


CheckButton
ColorPickerButton
MenuButton
OptionButton
Button
LinkButton
TextureButton
BaseButton
ColorRect
AspectRatioContainer
HBoxContainer
ColorPicker
VBoxContainer
BoxContainer
CenterContainer
HFlowContainer
VFlowContainer
FlowContainer
GraphNode
GraphElement
GridContainer
HSplitContainer
VSplitContainer
SplitContainer
MarginContainer
PanelContainer
ScrollContainer
SubViewportContainer
TabContainer
Container
GraphEdit
Control
ItemList
Label
LineEdit
MenuBar
NinePatchRect
Panel
ProgressBar
HScrollBar
ScrollBar
VScrollBar
Range
SpinBox
HSlider
Slider
VSlider
TextureProgress
ReferenceRect
RichTextLabel
HSeparator
Separator
VSeparator
TabBar
TextEdit
TextureRect
Tree
VideoStreamPlayer

Node2D


AnimatedSprite2D
AudioListener2D
AudioStreamPlayer2D
BackBufferCopy
Bone2D
Camera2D
CanvasGroup
CanvasModulate
Area2D
StaticBody2D
CharacterBody2D
PhysicalBone2D
RigidBody2D
PhysicsBody2D
CollisionObject2D
CollisionPolygon2D
CollisionShape2D
CPUParticles2D
GPUParticles2D
DampedSpringJoint2D
GrooveJoint2D
PinJoint2D
Joint2D
DirectionalLight2D
PointLight2D
Light2D
Node2D
LightOccluder2D
Line2D
Marker2D
MeshInstance2D
MultiMeshInstance2D
NavigationLink2D
NavigationObstacle2D
NavigationRegion2D
ParallaxLayer
Path2D
PathFollow2D
Polygon2D
RayCast2D
RemoteTransform2D
ShapeCast2D
Skeleton2D
Sprite2D
TileMap
TouchScreenButton
VisibleOnScreenNotifier2D
VisibleOnScreenEnabler2D

Node3D


AudioListener3D
AudioStreamPlayer3D
BoneAttachment3D
XRCamera3D
Camera3D
Area
CollisionObject3D
KinematicBody
RigidBody
StaticBody
VehicleBody
PhysicsBody
CollisionPolygon3D
CollisionShape3D
GridMap
ImporterMeshInstance3D
ConeTwistJoint3D
Generic6DOFJoint3D
HingeJoint3D
PinJoint3D
SliderJoint3D
Joint3D
LightmapProbe
Marker3D
Node3D
NavigationLink3D
NavigationObstacle 3D
NavigationRegion3D
OccluderInstance3D
OpenXRHand
Path3D
PathFollow3D
ProximityGroup
RayCast3D
RemoteTransform3D
ShapeCast3D
Skeleton3D
SpatialPlayer
SpatialSamplePlayer
SpatialStreamPlayer
SpringArm3D
VehicleWheel3D
XRNode3D
XRAnchor3D
XRController3D
XROrigin3D
Decal
FogVolume
GPUParticlesAttractorBox3D
GPUParticlesAttractorVectorField3D
GPUParticlesAttractorVectorField3D
GPUParticlesAttractor3D
GPUParticlesCollisionBox3D
GPUParticlesCollisionHeightField3D
GPUParticlesCollisionSDF3D
GPUParticleesCollisionSphere3D
GPUParticlesCollision3D
VisualInstance3D
Light3D
DirectionalLight3D
OmniLight3D
SpotLight3D
LightmapGI
ReflectionProbe
RootMotionView
VisibleOnScreenNotifier3D
VisibleOnScreenEnabler3D
VoxelGI
GeometryInstance3D
CPUParticles3D
CSGShape3D
GPUParticles3D
Label3D
MeshInstance3D
MultiMeshInstance3D
SpriteBase3D
AnimatedSprite3D
Sprite3D

Source files: **`class_tree.zip`**.

Internal rendering architecture

This page is a high-level overview of Godot 4's internal renderer design. It does not apply to previous Godot versions.

The goal of this page is to document design decisions taken to best suit Godot's design philosophy, while providing a starting point for new rendering contributors.

If you have questions about rendering internals not answered here, feel free to ask in the `#rendering` channel of the Godot Contributors Chat [https://chat.godotengine.org/channel/rendering].

Note

If you have difficulty understanding concepts on this page, it is recommended to go through an OpenGL tutorial such as LearnOpenGL [https://learnopengl.com/].

Modern low-level APIs (Vulkan/Direct3D 12) require intermediate knowledge of higher-level APIs (OpenGL/Direct3D 11) to be used effectively. Thankfully, contributors rarely need to work directly with low-level APIs. Godot's renderers are built entirely on OpenGL and RenderingDevice, which is our abstraction over Vulkan/Direct3D 12.

Rendering methods

Forward+

This is a forward renderer that uses a *clustered* approach to lighting.

Clustered lighting uses a compute shader to group lights into a 3D frustum aligned grid. Then, at render time, pixels can lookup what lights affect the grid cell they are in and only run light calculations for lights that might affect that pixel.

This approach can greatly speed up rendering performance on desktop hardware, but is substantially less efficient on mobile.

Forward Mobile

This is a forward renderer that uses a traditional single-pass approach to lighting.

Intended for mobile platforms, but can also run on desktop platforms. This rendering method is optimized to perform well on mobile GPUs. Mobile GPUs have a very different architecture compared to desktop GPUs due to their unique constraints around battery usage, heat, and overall bandwidth limitations of reading and writing data. Compute shaders also have very limited support or aren't supported at all. As a result, the mobile renderer purely uses raster-based shaders (fragment/vertex).

Unlike desktop GPUs, mobile GPUs perform *tile-based rendering*. Instead of rendering the whole image as a single unit, the image is divided in smaller tiles that fit within the faster internal memory of the mobile GPU. Each tile is rendered and then written out to the destination texture. This all happens automatically on the graphics driver.

The problem is that this introduces bottlenecks in our traditional approach. For desktop rendering, we render all opaque geometry, then handle the background, then transparent geometry, then post-processing. Each pass will need to read the current result into tile memory, perform its operations and then write it out again. We then wait for all tiles to be completed before moving on to the next pass.

The first important change in the mobile renderer is that the mobile renderer does not use the RGBA16F texture formats that the desktop renderer does. Instead, it is using an R10G10B10A2 UNORM texture format. This halves the bandwidth required and has further improvements as mobile hardware often further optimizes for 32-bit formats. The tradeoff is that the mobile renderer has limited HDR capabilities due to the reduced precision and maximum values in the color data.

The second important change is the use of sub-passes whenever possible. Sub-passes allows us to perform the rendering steps end-to-end per tile saving on the overhead introduced by reading from and writing to the tiles between each rendering pass. The ability to use sub-passes is limited by the inability to read neighboring pixels, as we're constrained to working within a single tile.

This limitation of subpasses results in not being able to implement features such as glow and depth of field efficiently. Similarly, if there is a requirement to read from the screen texture or depth texture, we must fully write out the rendering result limiting our ability to use sub-passes. When such features are enabled, a mix of sub-passes and normal passes are used, and these features result in a notable performance penalty.

On desktop platforms, the use of sub-passes won't have any impact on performance. However, this rendering method can still perform better than Clustered Forward in simple scenes thanks to its lower complexity and lower bandwidth usage. This is especially noticeable on low-end GPUs, integrated graphics or in VR applications.

Given its low-end focus, this rendering method does not provide high-end rendering features such as SDFGI and Volumetric fog and fog volumes. Several post-processing effects are also not available.

Compatibility

Note

This is the only rendering method available when using the OpenGL driver. This rendering method is not available when using Vulkan or Direct3D 12.

This is a traditional (non-clustered) forward renderer. It's intended for old GPUs that don't have Vulkan support, but still works very efficiently on newer hardware. Specifically, it is optimized for older and lower-end mobile devices However, many optimizations carry over making it a good choice for older and lower-end desktop as well.

Like the Mobile renderer, the Compatibility renderer uses an R10G10B10A2 UNORM texture for 3D rendering. Unlike the mobile renderer, colors are tonemapped and stored in sRGB format so there is no HDR support. This avoids the need for a tonemapping pass and allows us to use the lower bit texture without substantial banding.

The Compatibility renderer uses a traditional forward single-pass approach to drawing objects with lights, but it uses a multi-pass approach to draw lights with shadows. Specifically, in the first pass, it can draw multiple lights without shadows and up to one DirectionalLight3D with shadows. In each subsequent pass, it can draw up to one OmniLight3D, one SpotLight3D and one DirectionalLight3D with shadows. Lights with shadows will affect the scene differently than lights without shadows, as the lighting is blended in sRGB space instead of linear space. This difference in lighting will impact how the scene looks and needs to be kept in mind when designing scenes for the Compatibility renderer.

Given its low-end focus, this rendering method does not provide high-end rendering features (even less so compared to Forward

Mobile). Most post-processing effects are not available.

Why not deferred rendering?

Forward rendering generally provides a better tradeoff for performance versus flexibility, especially when a clustered approach to lighting is used. While deferred rendering can be faster in some cases, it's also less flexible and requires using hacks to be able to use MSAA. Since games with a less realistic art style can benefit a lot from MSAA, we chose to go with forward rendering for Godot 4 (like Godot 3).

That said, parts of the forward renderer *are* performed with a deferred approach to allow for some optimizations when possible. This applies to VoxelGI and SDFGI in particular.

A clustered deferred renderer may be developed in the future. This renderer could be used in situations where performance is favored over flexibility.

Rendering drivers

Godot 4 supports the following graphics APIs:

Vulkan

This is the main driver in Godot 4, with most of the development focus going towards this driver.

Vulkan 1.0 is required as a baseline, with optional Vulkan 1.1 and 1.2 features used when available. volk [https://github.com/zeux/volk] is used as a Vulkan loader, and Vulkan Memory Allocator [https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator] is used for memory management.

Both the Forward+ and Mobile Rendering methods are supported when using the Vulkan driver.

Vulkan context creation:

- drivers/vulkan/vulkan_context.cpp [https://github.com/godotengine/godot/blob/4.2/drivers/vulkan/vulkan_context.cpp]

Direct3D 12 context creation:

- drivers/d3d12/d3d12_context.cpp [https://github.com/godotengine/godot/blob/master/drivers/d3d12/d3d12_context.cpp]

Direct3D 12

Like Vulkan, the Direct3D 12 driver targets modern platforms only. It is designed to target both Windows and Xbox (whereas Vulkan can't be used directly on Xbox).

Both the Forward+ and Mobile Rendering methods can be used with Direct3D 12.

Core shaders are shared with the Vulkan renderer. Shaders are transpiled from SPIR-V to DXIL using Mesa NIR (more information [https://godotengine.org/article/d3d12-adventures-in-shaderland/]).

This driver is still experimental and only available in Godot 4.3 and later. While Direct3D 12 allows supporting Direct3D-exclusive features on Windows 11 such as windowed optimizations and Auto HDR, Vulkan is still recommended for most projects. See the pull request that introduced Direct3D 12 support [https://github.com/godotengine/godot/pull/70315] for more information.

Metal

Godot supports Metal rendering via MoltenVK [https://github.com/KhronosGroup/MoltenVK], as macOS and iOS do not support Vulkan natively. This is done automatically when specifying the Vulkan driver in the Project Settings.

MoltenVK makes driver maintenance easy at the cost of some performance overhead. Also, MoltenVK has several limitations that a native Metal driver implementation wouldn't have. Both the clustered and mobile Rendering methods can be used with a Metal backend via MoltenVK.

A native Metal driver is planned in the future for better performance and compatibility.

OpenGL

This driver uses OpenGL ES 3.0 and targets legacy and low-end devices that don't support Vulkan. OpenGL 3.3 Core Profile is used on desktop platforms to run this driver, as most graphics drivers on desktop don't support OpenGL ES. WebGL 2.0 is used for web exports.

It is possible to use the use of OpenGL ES 3.0 directly on desktop platforms using the `--rendering-driver opengl3_es` command line argument, although this will only work on graphics drivers that feature native OpenGL ES support (such as Mesa).

Only the Compatibility rendering method can be used with the OpenGL driver.

Core shaders are entirely different from the Vulkan renderer.

Many advanced features are not supported with this driver, as it targets low-end devices first and foremost.

Summary of rendering drivers/methods

The following rendering API + rendering method combinations are currently possible:

- Vulkan + Forward+
- Vulkan + Forward Mobile
- Direct3D 12 + Forward+
- Direct3D 12 + Forward Mobile
- Metal + Forward+ (via MoltenVK)
- Metal + Forward Mobile (via MoltenVK)
- OpenGL + Compatibility

Each combination has its own limitations and performance characteristics. Make sure to test your changes on all rendering methods if possible before opening a pull request.

RenderingDevice abstraction

Note

The OpenGL driver does not use the RenderingDevice abstraction.

To make the complexity of modern low-level graphics APIs more manageable, Godot uses its own abstraction called RenderingDevice.

This means that when writing code for modern rendering methods, you don't actually use the Vulkan or Direct3D 12 APIs directly. While this is still lower-level than an API like OpenGL, this makes working on the renderer easier, as RenderingDevice will abstract many API-specific quirks for you. The RenderingDevice presents a similar level of abstraction as Metal or WebGPU.

Vulkan RenderingDevice implementation:

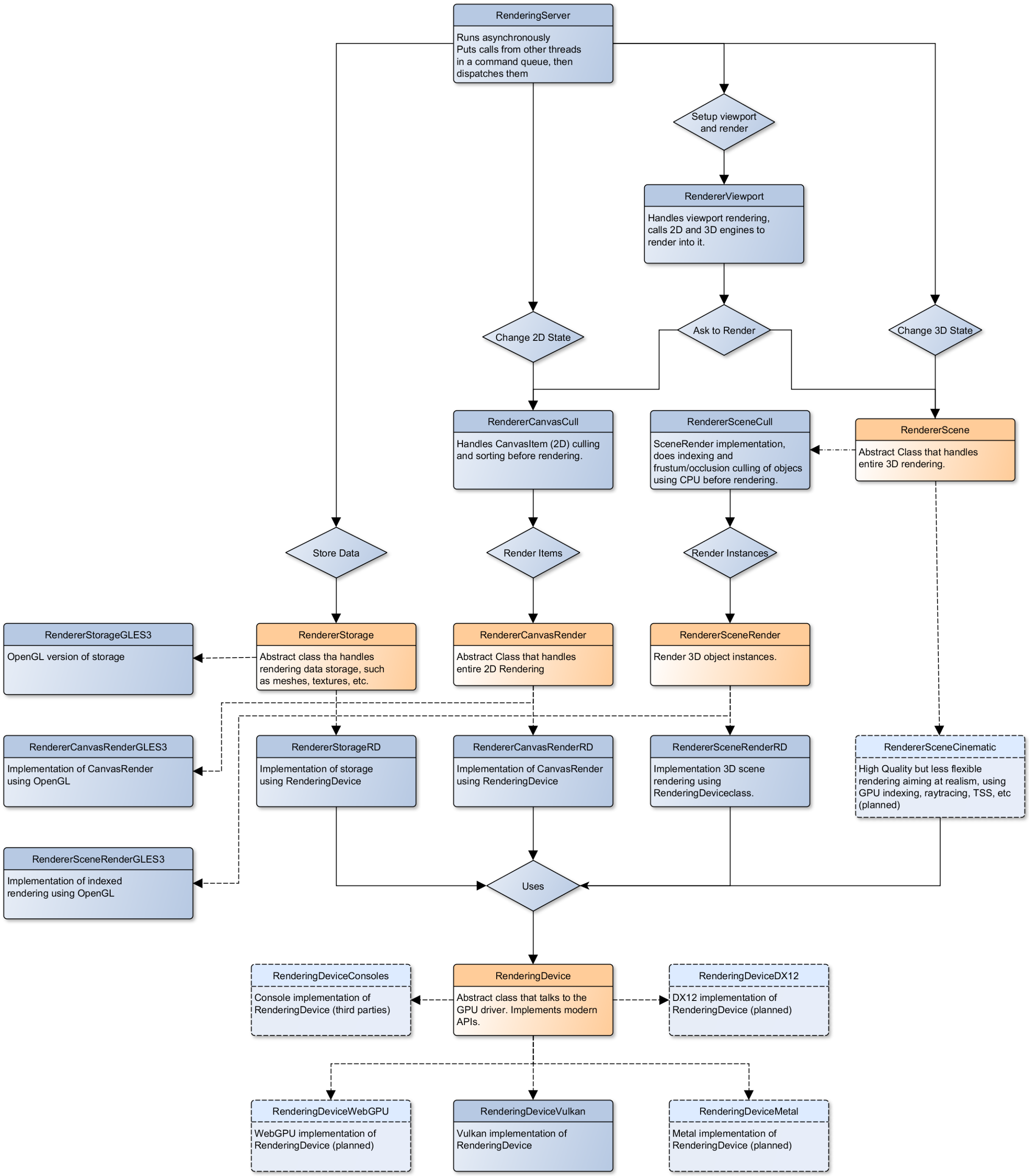
- drivers/vulkan/rendering_device_driver_vulkan.cpp [https://github.com/godotengine/godot/blob/master/drivers/vulkan/rendering_device_driver_vulkan.cpp]

Direct3D 12 RenderingDevice implementation:

- drivers/d3d12/rendering_device_driver_d3d12.cpp [https://github.com/godotengine/godot/blob/master/drivers/d3d12/rendering_device_driver_d3d12.cpp]

Core rendering classes architecture

This diagram represents the structure of rendering classes in Godot, including the RenderingDevice abstraction:


RenderingServer
Runs asynchronously
Puts calls from other threads
in a command queue, then
dispatches them
Setup viewport
and render
RendererViewport
Handles viewport rendering,
calls 2D and 3D engines to
render into it.
Change 2D State
Ask to Render
Change 3D State
RendererCanvasCull
Handles CanvasItem (2D) culling
and sorting before rendering.
RendererSceneCull
SceneRender implementation,
does indexing and
frustum/occlusion culling of objecs
using CPU before rendering.
RendererScene
Abstract Class that handles
entire 3D rendering.
Store Data
Render Items
Render Instances
RendererStorageGLES3
OpenGL version of storage
RendererStorage
Abstract class tha handles
rendering data storage, such
as meshes, textures, etc.
RendererCanvasRender
Abstract Class that handles
entire 2D Rendering
RendererSceneRender
Render 3D object instances.
RendererCanvasRenderGLES3
Implementation of CanvasRender
using OpenGL
RendererStorageRD
Implementation of storage
using RenderingDevice
RendererCanvasRenderRD
Implementation of CanvasRender
using RenderingDevice
RendererSceneRenderRD
Implementation 3D scene
rendering using
RenderingDeviceclass.
RendererSceneCinematic
High Quality but less flexible
rendering aiming at realism, using
GPU indexing, raytracing, TSS, etc
(planned)
RendererSceneRenderGLES3
Implementation of indexed
rendering using OpenGL
Uses
RenderingDeviceConsoles
Console implementation of
RenderingDevice (third parties)
RenderingDevice
Abstract class that talks to the
GPU driver. Implements modern
APIs.
RenderingDeviceDX12
DX12 implementation of
RenderingDevice (planned)
RenderingDeviceWebGPU
WebGPU implementation of
RenderingDevice (planned)
RenderingDeviceVulkan
Vulkan implementation of
RenderingDevice
RenderingDeviceMetal
Metal implementation of
RenderingDevice (planned)

View at full size [https://raw.githubusercontent.com/godotengine/godot-docs/master/contributing/development/core_and_modules/img/rendering_architecture_diagram.webp]

Core shaders

While shaders in Godot projects are written using a custom language inspired by GLSL, core shaders are written directly in GLSL.

These core shaders are embedded in the editor and export template binaries at compile-time. To see any changes you've made to those GLSL shaders, you need to recompile the editor or export template binary.

Some material features such as height mapping, refraction and proximity fade are not part of core shaders, and are performed in the default BaseMaterial3D using the Godot shader language instead (not GLSL). This is done by procedurally generating the required shader code depending on the features enabled in the material.

By convention, shader files with `_inc` in their name are included in other GLSL files for better code reuse. Standard GLSL preprocessing is used to achieve this.

Warning

Core material shaders will be used by every material in the scene – both with the default BaseMaterial3D and custom shaders. As a result, these shaders must be kept as simple as possible to avoid performance issues and ensure shader compilation doesn't become too slow.

If you use `if` branching in a shader, performance may decrease as VGPR usage will increase in the shader. This happens even if all pixels evaluate to `true` or `false` in a given frame.

If you use `#if` preprocessor branching, the number of required shader versions will increase in the scene. In a worst-case scenario, adding a single boolean `#define` can *double* the number of shader versions that may need to be compiled in a given scene. In some cases, Vulkan specialization constants can be used as a faster (but more limited) alternative.

This means there is a high barrier to adding new built-in material features in Godot, both in the core shaders and BaseMaterial3D. While BaseMaterial3D can make use of dynamic code generation to only include the shader code if the feature is enabled, it'll still require generating more shader versions when these features are used in a project. This can make shader compilation stutter more noticeable in complex 3D scenes.

See The Shader Permutation Problem [https://therealmjp.github.io/posts/shader-permutations-part1/] and Branching on a GPU [https://medium.com/@jasonbooth_86226/branching-on-a-gpu-18bfc83694f2] blog posts for more information.

Core GLSL material shaders:

- Forward+: servers/rendering/renderer_rd/shaders/forward_clustered/scene_forward_clustered.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/forward_clustered/scene_forward_clustered.glsl]
- Forward Mobile: servers/rendering/renderer_rd/shaders/forward_mobile/scene_forward_mobile.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/forward_mobile/scene_forward_mobile.glsl]
- Compatibility: drivers/gles3/shaders/scene.glsl [https://github.com/godotengine/godot/blob/4.2/drivers/gles3/shaders/scene.glsl]

Material shader generation:

- scene/resources/material.cpp [https://github.com/godotengine/godot/blob/4.2/scene/resources/material.cpp]

Other GLSL shaders for Forward+ and Forward Mobile rendering methods:

- servers/rendering/renderer_rd/shaders/ [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/]
- modules/lightmapper_rd/ [https://github.com/godotengine/godot/blob/4.2/modules/lightmapper_rd]

Other GLSL shaders for the Compatibility rendering method:

- drivers/gles3/shaders/ [https://github.com/godotengine/godot/blob/4.2/drivers/gles3/shaders/]

2D and 3D rendering separation

Note

The following is only applicable in the Forward+ and Forward Mobile rendering methods, not in Compatibility. Multiple Viewports can be used to emulate this when using the Compatibility backend, or to perform 2D resolution scaling.

2D and 3D are rendered to separate buffers, as 2D rendering in Godot is performed in LDR sRGB-space while 3D rendering uses HDR linear space.

The color format used for 2D rendering is RGB8 (RGBA8 if the **Transparent** property on the Viewport is enabled). 3D rendering uses a 24-bit unsigned normalized integer depth buffer, or 32-bit signed floating-point if a 24-bit depth buffer is not supported by the hardware. 2D rendering does not use a depth buffer.

3D resolution scaling is performed differently depending on whether bilinear or FSR 1.0 scaling is used. When bilinear scaling is used, no special upscaling shader is run. Instead, the viewport's texture is stretched and displayed with a linear sampler (which makes the filtering happen directly on the hardware). This allows maximizing the performance of bilinear 3D scaling.

The `configure()` function in RenderSceneBuffersRD reallocates the 2D/3D buffers when the resolution or scaling changes.

Dynamic resolution scaling isn't supported yet, but is planned in a future Godot release.

2D and 3D rendering buffer configuration C++ code:

- servers/rendering/renderer_rd/storage_rd/render_scene_buffers_rd.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/storage_rd/render_scene_buffers_rd.cpp]

FSR 1.0:

- servers/rendering/renderer_rd/effects/fsr.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/effects/fsr.cpp]
- thirdparty/amd-fsr/ [https://github.com/godotengine/godot/tree/master/thirdparty/amd-fsr]

2D rendering techniques

2D light rendering is performed in a single pass to allow for better performance with large amounts of lights.

The Forward+ and Mobile rendering methods don't feature 2D batching yet, but it's planned for a future release.

The Compatibility backend features 2D batching to improve performance, which is especially noticeable with lots of text on screen.

MSAA can be enabled in 2D to provide "automatic" line and polygon antialiasing, but FXAA does not affect 2D rendering as it's calculated before 2D rendering begins. Godot's 2D drawing methods such as the Line2D node or some CanvasItem `draw_*()` methods provide their own way of antialiasing based on triangle strips and vertex colors, which don't require MSAA to work.

A 2D signed distance field representing LightOccluder2D nodes in the viewport is automatically generated if a user shader requests it. This can be used for various effects in custom shaders, such as 2D global illumination. It is also used to calculate particle collisions in 2D.

2D SDF generation GLSL shader:

- servers/rendering/renderer_rd/shaders/canvas_sdf.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/canvas_sdf.glsl]

3D rendering techniques

Batching and instancing

In the Forward+ backend, Vulkan instancing is used to group rendering of identical objects for performance. This is not as fast as static mesh merging, but it still allows instances to be culled individually.

Light, decal and reflection probe rendering

Note

Reflection probe and decal rendering are currently not available in the Compatibility backend.

As its name implies, the Forward+ backend uses clustered lighting. This allows using as many lights as you want; performance largely depends on screen coverage. Shadow-less lights can be almost free if they don't occupy much space on screen.

All rendering methods also support rendering up to 8 directional lights at the same time (albeit with lower shadow quality when more than one light has shadows enabled).

The Forward Mobile backend uses a single-pass lighting approach, with a limitation of 8 OmniLights + 8 SpotLights affecting each Mesh *resource* (plus a limitation of 256 OmniLights + 256 SpotLights in the camera view). These limits are hardcoded and can't be adjusted in the project settings.

The Compatibility backend uses a hybrid single-pass + multi-pass lighting approach. Lights without shadows are rendered in a single pass. Lights with shadows are rendered in multiple passes. This is required for performance reasons on mobile devices. As a result, performance does not scale well with many shadow-casting lights. It is recommended to only have a handful of lights with shadows in the camera frustum at a time and for those lights to be spread apart so that each object is only touched by 1 or 2 shadowed lights at a time. The maximum number of lights visible at once can be adjusted in the project settings.

In all 3 methods, lights without shadows are much cheaper than lights with shadows. To improve performance, lights are only updated when the light is modified or when objects in its radius are modified. Godot currently doesn't separate static shadow rendering from dynamic shadow rendering, but this is planned in a future release.

Clustering is also used for reflection probes and decal rendering in the Forward+ backend.

Shadow mapping

Both Forward+ and Forward Mobile methods use PCF to filter shadow maps and create a soft penumbra. Instead of using a fixed PCF pattern, these methods use a vogel disk pattern which allows for changing the number of samples and smoothly changing the quality.

Godot also supports percentage-closer soft shadows (PCSS) for more realistic shadow penumbra rendering. PCSS shadows are limited to the Forward+ backend as they're too demanding to be usable in the Forward Mobile backend. PCSS also uses a vogel-disk shaped kernel.

Additionally, both shadow-mapping techniques rotate the kernel on a per-pixel basis to help soften under-sampling artifacts.

The Compatibility backend doesn't support shadow mapping for any light types yet.

Temporal antialiasing

Note

Only available in the Forward+ backend, not the Forward Mobile or Compatibility methods.

Godot uses a custom TAA implementation based on the old TAA implementation from Spartan Engine [https://github.com/PanosK92/SpartanEngine].

Temporal antialiasing requires motion vectors to work. If motion vectors are not correctly generated, ghosting will occur when the camera or objects move.

Motion vectors are generated on the GPU in the main material shader. This is done by running the vertex shader corresponding to the previous rendered frame (with the previous camera transform) in addition to the vertex shader for the current rendered frame, then storing the difference between them in a color buffer.

Alternatively, FSR 2.2 can be used as an upscaling solution that also provides its own temporal antialiasing algorithm. FSR 2.2 is implemented on top of the RenderingDevice abstraction as opposed to using AMD's reference code directly.

TAA resolve:

- servers/rendering/renderer_rd/shaders/effects/taa_resolve.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/taa_resolve.glsl]

FSR 2.2:

- servers/rendering/renderer_rd/effects/fsr2.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/effects/fsr2.cpp]
- servers/rendering/renderer_rd/shaders/effects/fsr2/ [https://github.com/godotengine/godot/tree/master/servers/rendering/renderer_rd/shaders/effects/fsr2]
- thirdparty/amd-fsr2/ [https://github.com/godotengine/godot/tree/master/thirdparty/amd-fsr2]

Global illumination

Note

VoxelGI and SDFGI are only available in the Forward+ backend, not the Forward Mobile or Compatibility methods.

LightmapGI *baking* is only available in the Forward+ and Forward Mobile methods, and can only be performed within the editor (not in an exported project). LightmapGI *rendering* will eventually be supported by the Compatibility backend.

Godot supports voxel-based GI (VoxelGI), signed distance field GI (SDFGI) and lightmap baking and rendering (LightmapGI). These techniques can be used simultaneously if desired.

Lightmap baking happens on the GPU using Vulkan compute shaders. The GPU-based lightmapper is implemented in the LightmapperRD class, which inherits from the Lightmapper class. This allows for implementing additional lightmappers, paving the way for a future port of the CPU-based lightmapper present in Godot 3.x. This would allow baking lightmaps while using the Compatibility backend.

Core GI C++ code:

- servers/rendering/renderer_rd/environment/gi.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/environment/gi.cpp]
- scene/3d/voxel_gi.cpp [https://github.com/godotengine/godot/blob/4.2/scene/3d/voxel_gi.cpp] - VoxelGI node
- editor/plugins/voxel_gi_editor_plugin.cpp [https://github.com/godotengine/godot/blob/4.2/editor/plugins/voxel_gi_editor_plugin.cpp] - Editor UI for the VoxelGI node

Core GI GLSL shaders:

- servers/rendering/renderer_rd/shaders/environment/voxel_gi.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/voxel_gi.glsl]
- servers/rendering/renderer_rd/shaders/environment/voxel_gi_debug.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/voxel_gi_debug.glsl] - VoxelGI debug draw mode
- servers/rendering/renderer_rd/shaders/environment/sdfgi_debug.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/sdfgi_debug.glsl] - SDFGI Cascades debug draw mode
- servers/rendering/renderer_rd/shaders/environment/sdfgi_debug_probes.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/sdfgi_debug_probes.glsl] - SDFGI Probes debug draw mode
- servers/rendering/renderer_rd/shaders/environment/sdfgi_integrate.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/sdfgi_integrate.glsl]
- servers/rendering/renderer_rd/shaders/environment/sdfgi_preprocess.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/sdfgi_preprocess.glsl]
- servers/rendering/renderer_rd/shaders/environment/sdfgi_direct_light.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/sdfgi_direct_light.glsl]

Lightmapper C++ code:

- scene/3d/lightmap_gi.cpp [https://github.com/godotengine/godot/blob/4.2/scene/3d/lightmap_gi.cpp]

- LightmapGI node
- editor/plugins/lightmap_gi_editor_plugin.cpp [https://github.com/godotengine/godot/blob/4.2/editor/plugins/lightmap_gi_editor_plugin.cpp] - Editor UI for the LightmapGI node
- scene/3d/lightmapper.cpp [https://github.com/godotengine/godot/blob/4.2/scene/3d/lightmapper.cpp] - Abstract class
- modules/lightmapper_rd/lightmapper_rd.cpp [https://github.com/godotengine/godot/blob/4.2/modules/lightmapper_rd/lightmapper_rd.cpp] - GPU-based lightmapper implementation

Lightmapper GLSL shaders:

- modules/lightmapper_rd/lm_raster.glsl [https://github.com/godotengine/godot/blob/4.2/modules/lightmapper_rd/lm_raster.glsl]
- modules/lightmapper_rd/lm_compute.glsl [https://github.com/godotengine/godot/blob/4.2/modules/lightmapper_rd/lm_compute.glsl]
- modules/lightmapper_rd/lm_blendseams.glsl [https://github.com/godotengine/godot/blob/4.2/modules/lightmapper_rd/lm_blendseams.glsl]

Depth of field

Note

Only available in the Forward+ and Forward Mobile methods, not the Compatibility backend.

The Forward+ and Forward Mobile methods use different approaches to DOF rendering, with different visual results. This is done to best match the performance characteristics of the target hardware. In Clustered Forward, DOF is performed using a compute shader. In Forward Mobile, DOF is performed using a fragment shader (raster).

Box, hexagon and circle bokeh shapes are available (from fastest to slowest). Depth of field can optionally be jittered every frame to improve its appearance when temporal antialiasing is enabled.

Depth of field C++ code:

- servers/rendering/renderer_rd/effects/bokeh_dof.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/effects/bokeh_dof.cpp]

Depth of field GLSL shader (compute - used for Forward+):

- servers/rendering/renderer_rd/shaders/effects/bokeh_dof.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/bokeh_dof.glsl]

Depth of field GLSL shader (raster - used for Forward Mobile):

- servers/rendering/renderer_rd/shaders/effects/bokeh_dof_raster.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/bokeh_dof_raster.glsl]

Screen-space effects (SSAO, SSIL, SSR, SSS)

Note

Only available in the Forward+ backend, not the Forward Mobile or Compatibility methods.

The Forward+ backend supports screen-space ambient occlusion, screen-space indirect lighting, screen-space reflections and subsurface scattering.

SSAO uses an implementation derived from Intel's ASSAO [https://www.intel.com/content/www/us/en/developer/articles/technical/adaptive-screen-space-ambient-occlusion.html] (converted to Vulkan). SSIL is derived from SSAO to provide high-performance indirect lighting.

When both SSAO and SSIL are enabled, parts of SSAO and SSIL are shared to reduce the performance impact.

SSAO and SSIL are performed at half resolution by default to improve performance. SSR is always performed at half resolution to improve performance.

Screen-space effects C++ code:

- servers/rendering/renderer_rd/effects/ss_effects.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/effects/ss_effects.cpp]

Screen-space ambient occlusion GLSL shader:

- servers/rendering/renderer_rd/shaders/effects/ssao.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssao.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssao_blur.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssao_blur.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssao_interleave.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssao_interleave.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssao_importance_map.glsl

[https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssao_importance_map.glsl]

Screen-space indirect lighting GLSL shader:

- servers/rendering/renderer_rd/shaders/effects/ssil.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssil.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssil_blur.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssil_blur.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssil_interleave.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssil_interleave.glsl]
- servers/rendering/renderer_rd/shaders/effects/ssil_importance_map.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/ssil_importance_map.glsl]

Screen-space reflections GLSL shader:

- servers/rendering/renderer_rd/shaders/effects/screen_space_reflection.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/screen_space_reflection.glsl]
- servers/rendering/renderer_rd/shaders/effects/screen_space_reflection_scale.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/screen_space_reflection_scale.glsl]
- servers/rendering/renderer_rd/shaders/effects/screen_space_reflection_filter.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/screen_space_reflection_filter.glsl]

Subsurface scattering GLSL:

- servers/rendering/renderer_rd/shaders/effects/subsurface_scattering.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/effects/subsurface_scattering.glsl]

Sky rendering

See also

Sky shaders

Godot supports using shaders to render the sky background. The radiance map (which is used to provide ambient light and reflections for PBR materials) is automatically updated based on the sky shader.

The SkyMaterial resources such as ProceduralSkyMaterial, PhysicalSkyMaterial and PanoramaSkyMaterial generate a built-in shader for sky rendering. This is similar to what BaseMaterial3D provides for 3D scene materials.

A detailed technical implementation can be found in the Custom sky shaders in Godot 4.0 [https://godotengine.org/article/custom-sky-shaders-godot-4-0] article.

Sky rendering C++ code:

- servers/rendering/renderer_rd/environment/sky.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/environment/sky.cpp] - Sky rendering
- scene/resources/sky.cpp [https://github.com/godotengine/godot/blob/4.2/scene/resources/sky.cpp] - Sky resource (not to be confused with sky rendering)
- scene/resources/sky_material.cpp [https://github.com/godotengine/godot/blob/4.2/scene/resources/sky_material.cpp] SkyMaterial resources (used in the Sky resource)

Sky rendering GLSL shader:

Volumetric fog

Note

Only available in the Forward+ backend, not the Forward Mobile or Compatibility methods.

See also

Fog shaders

Godot supports a frustum-aligned voxel (froxel) approach to volumetric fog rendering. As opposed to a post-processing filter, this approach is more general-purpose as it can work with any light type. Fog can also use shaders for custom behavior, which allows animating the fog or using a 3D texture to represent density.

The FogMaterial resource generates a built-in shader for FogVolume nodes. This is similar to what BaseMaterial3D provides for 3D scene materials.

A detailed technical explanation can be found in the Fog Volumes arrive in Godot 4.0 [https://godotengine.org/article/fog-volumes-arrive-in-godot-4] article.

Volumetric fog C++ code:

- servers/rendering/renderer_rd/environment/fog.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/environment/fog.cpp] - General volumetric fog
- scene/3d/fog_volume.cpp [https://github.com/godotengine/godot/blob/4.2/scene/3d/fog_volume.cpp]

- FogVolume node
- [scene/resources/fog_material.cpp](https://github.com/godotengine/godot/blob/4.2/scene/resources/fog_material.cpp) [https://github.com/godotengine/godot/blob/4.2/scene/resources/fog_material.cpp] - FogMaterial resource (used by FogVolume)

Volumetric fog GLSL shaders:

- servers/rendering/renderer_rd/shaders/environment/volumetric_fog.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/volumetric_fog.glsl]
- servers/rendering/renderer_rd/shaders/environment/volumetric_fog_process.glsl [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_rd/shaders/environment/volumetric_fog_process.glsl]

Occlusion culling

While modern GPUs can handle drawing a lot of triangles, the number of draw calls in complex scenes can still be a bottleneck (even with Vulkan and Direct3D 12).

Godot 4 supports occlusion culling to reduce overdraw (when the depth prepass is disabled) and reduce vertex throughput. This is done by rasterizing a low-resolution buffer on the CPU using Embree [https://github.com/embree/embree]. The buffer's resolution depends on the number of CPU threads on the system, as this is done in parallel. This buffer includes occluder shapes that were baked in the editor or created at run-time.

As complex occluders can introduce a lot of strain on the CPU, baked occluders can be simplified automatically when generated in the editor.

Godot's occlusion culling doesn't support dynamic occluders yet, but OccluderInstance3D nodes can still have their visibility toggled or be moved. However, this will be slow when updating complex occluders this way. Therefore, updating occluders at run-time is best done only on simple occluder shapes such as quads or cuboids.

This CPU-based approach has a few advantages over other solutions, such as portals and rooms or a GPU-based culling solution:

- No manual setup required (but can be tweaked manually for best performance).
- No frame delay, which is problematic in cutscenes during camera cuts or when the camera moves fast behind a wall.
- Works the same on all rendering drivers and methods, with no unpredictable behavior depending on the driver or GPU hardware.

Occlusion culling is performed by registering occluder meshes, which is done using OccluderInstance3D *nodes* (which themselves use Occluder3D *resources*). RenderingServer then performs occlusion culling by calling Embree in RendererSceneOcclusionCull.

Occlusion culling C++ code:

- scene/3d/occluder_instance_3d.cpp [https://github.com/godotengine/godot/blob/4.2/scene/3d/occluder_instance_3d.cpp]
- servers/rendering/renderer_scene_occlusion_cull.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_scene_occlusion_cull.cpp]

Visibility range (LOD)

Godot supports manually authored hierarchical level of detail (HLOD), with distances specified by the user in the inspector.

In RenderingSceneCull, the `_scene_cull()` and `_render_scene()` functions are where most of the LOD determination happens. Each viewport can render the same mesh with different LODs (to allow for split screen rendering to look correct).

Visibility range C++ code:

- servers/rendering/renderer_scene_cull.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_scene_cull.cpp]

Automatic mesh LOD

The ImporterMesh class is used for the 3D mesh import workflow in the editor. Its `generate_lods()` function handles generating using the meshoptimizer [https://meshoptimizer.org/] library.

LOD mesh generation also generates shadow meshes at the same time. These are meshes that have their vertices welded regardless of smoothing and materials. This is used to improve shadow rendering performance by lowering the vertex throughput required to render shadows.

The RenderingSceneCull class's `_render_scene()` function determines which mesh LOD should be used when rendering. Each viewport can render the same mesh with different LODs (to allow for split screen rendering to look correct).

The mesh LOD is automatically chosen based on a screen coverage metric. This takes resolution and camera FOV changes into account without requiring user intervention. The threshold multiplier can be adjusted in the project settings.

To improve performance, shadow rendering and reflection probe rendering also choose their own mesh LOD thresholds (which can be different from the main scene rendering).

Mesh LOD generation on import C++ code:

- scene/resources/importer_mesh.cpp [https://github.com/godotengine/godot/blob/4.2/scene/resources/importer_mesh.cpp]

Mesh LOD determination C++ code:

- servers/rendering/renderer_scene_cull.cpp [https://github.com/godotengine/godot/blob/4.2/servers/rendering/renderer_scene_cull.cpp]

2D coordinate systems and 2D transforms

Introduction

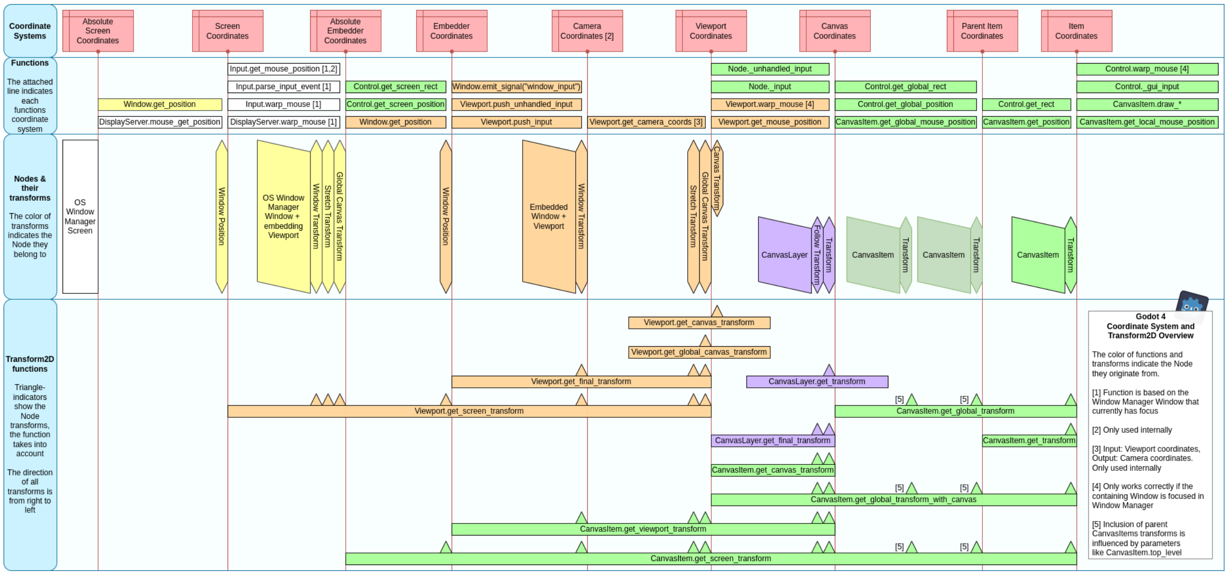
This is a detailed overview of the available 2D coordinate systems and 2D transforms that are built in. The basic concepts are covered in Viewport and canvas transforms.

Transform2D are matrices that convert coordinates from one coordinate system to an other. In order to use them, it is beneficial to know which coordinate systems are available in Godot. For a deeper understanding, the Matrices and transforms tutorial offers insights to the underlying functionality.

Godot 2D coordinate systems

The following graphic gives an overview of Godot 2D coordinate systems and the available node-transforms, transform-functions and coordinate-system related functions. At the left is the OS Window Manager screen, at the right are the CanvasItems. For simplicity reasons this graphic doesn't include SubViewport, SubViewportContainer, ParallaxLayer and ParallaxBackground all of which also influence transforms.

The graphic is based on a node tree of the following form: `Root Window (embed Windows)` ⇒ `Window (don't embed Windows)` ⇒ `CanvasLayer` ⇒ `CanvasItem` ⇒ `CanvasItem` ⇒ `CanvasItem`. There are more complex combinations possible, like deeply nested Window and SubViewports, however this example intends to provide an overview of the methodology in general.


Coordinate Systems
Absolute Screen Coordinates
Screen Coordinates
Absolute Embedder Coordinates
Embedder Coordinates
Camera Coordinates [2]
Viewport Coordinates
Canvas Coordinates
Parent Item Coordinates
Item Coordinates
Functions
The attached line indicates each functions coordinate system
Input.get_mouse_position [1,2]
Input.parse_input_event [1]
Input.warp_mouse [1]
DisplayServer.warp_mouse [1]
Window.get_position
DisplayServer.mouse_get_position
Control.get_screen_rect
Control.get_screen_position
Window.get_position
Window.emit_signal("window_input")
Viewport.push_unhandled_input
Viewport.push_input
Viewport.get_camera_coords [3]
Node._unhandled_input
Node._input
Viewport.warp_mouse [4]
Viewport.get_mouse_position
Control.get_global_rect
Control.get_global_position
CanvasItem.get_global_mouse_position
Control.get_rect
CanvasItem.get_position
Control.warp_mouse [4]
Control._gui_input
CanvasItem.draw_*
CanvasItem.get_local_mouse_position
Nodes & their transforms
The color of transforms indicates the Node they belong to
OS Window Manager Screen
Window Position
OS Window Manager Window + embedding Viewport
Window Transform
Stretch Transform
Global Canvas Transform
Window Position
Embedded Window + Viewport
Window Transform
Stretch Transform
Global Canvas Transform
Canvas Transform
CanvasLayer
Follow Transform
Transform
CanvasItem
Transform
CanvasItem
Transform
CanvasItem
Transform
Transform2D functions
Triangle-indicators show the Node transforms, the function takes into account
The direction of all transforms is from right to left
Viewport.get_canvas_transform
Viewport.get_global_canvas_transform
Viewport.get_final_transform
CanvasLayer.get_transform
Viewport.get_screen_transform
[5]
[5]
CanvasItem.get_global_transform
CanvasLayer.get_final_transform
CanvasItem.get_transform
CanvasItem.get_canvas_transform
[5]
[5]
CanvasItem.get_global_transform_with_canvas
CanvasItem.get_viewport_transform
[5]
[5]
CanvasItem.get_screen_transform

Godot 4
Coordinate System and Transform2D Overview

The color of functions and transforms indicate the Node they originate from.

[1] Function is based on the Window Manager Window that currently has focus

[2] Only used internally

[3] Input: Viewport coordinates, Output: Camera coordinates. Only used internally

[4] Only works correctly if the containing Window is focused in Window Manager

[5] Inclusion of parent CanvasItems transforms is influenced by parameters like CanvasItem.top_level

Click graphic to enlarge.

- **Item Coordinates**
  This is the local coordinate system of a CanvasItem.

- **Parent Item Coordinates**
  This is the local coordinate system of the parent's *CanvasItem*. When positioning *CanvasItems* in the *Canvas*, they usually inherit the transformations of their parent *CanvasItems*. An exceptions is CanvasItems.top_level.

- **Canvas Coordinates**
  As mentioned in the previous tutorial Canvas layers, there are two types of canvases (*Viewport* canvas and *CanvasLayer* canvas) and both have a canvas coordinate system. These are also called world coordinates. A *Viewport* can contain multiple *Canvases* with different coordinate systems.

- **Viewport Coordinates**
  This is the coordinate system of the Viewport.

- **Camera Coordinates**

This is only used internally for functionality like 3D-camera ray projections.

- **Embedder Coordinates / Screen Coordinates**

  Every *Viewport* (*Window* or *SubViewport*) in the scene tree is embedded either in a different node or in the OS Window Manager. This coordinate system's origin is identical to the top-left corner of the *Window* or *SubViewport* and its scale is the one of the embedder or the OS Window Manager.

  If the embedder is the OS Window Manager, then they are also called Screen Coordinates.

- **Absolute Embedder Coordinates / Absolute Screen Coordinates**

  The origin of this coordinate system is the top-left corner of the embedding node or the OS Window Manager screen. Its scale is the one of the embedder or the OS Window Manager.

  If the embedder is the OS Window Manager, then they are also called Absolute Screen Coordinates.

Node transforms

Each of the mentioned nodes have one or more transforms associated with them and the combination of these nodes infer the transforms between the different coordinate systems. With a few exceptions, the transforms are Transform2D and the following list shows details and effects of each of them.

- **CanvasItem transform**

  CanvasItems are either *Control*-nodes or *Node2D*-nodes.

  For *Control* nodes this transform consists of a position relative to the parent's origin and a scale and rotation around a pivot point.

For *Node2D* nodes transform consists of position, rotation, scale and skew.

The transform affects the item itself and usually also child-*CanvasItems* and in the case of a *SubViewportContainer* it affects the contained *SubViewport*.

- **CanvasLayer transform**
  The *CanvasLayer's* transform affects all *CanvasItems* within the *CanvasLayer*. It doesn't affect other *CanvasLayers* or *Windows* in its *Viewport*.

- **CanvasLayer follow viewport transform**
  The *follow viewport transform* is an automatically calculated transform, that is based on the *Viewport's* canvas transform and the *CanvasLayer's* follow viewport scale and can be used, if enabled, to achieve a pseudo 3D effect. It affects the same child nodes as the *CanvasLayer transform*.

- **Viewport canvas transform**
  The canvas transform affects all *CanvasItems* in the *Viewport's* default canvas. It also affects *CanvasLayers*, that have follow viewport transform enabled. The *Viewport's* active Camera2D works by changing this transform. It doesn't affect this *Viewport's* embedded *Windows*.

- **Viewport global canvas transform**
  Viewports also have a global canvas transform. This is the master transform and affects all individual *Canvas Layer* and embedded *Window* transforms. This is primarily used in Godot's CanvasItem Editor.

- **Viewport stretch transform**
  Finally, *Viewports* have a *stretch transform*, which is used when resizing or stretching the viewport. This transform is used for Windows as described in Multiple resolutions, but

can also be manually set on *SubViewports* by means of size and size_2d_override. It's translation, rotation and skew are the default values and it can only have non-default scale.

- **Window transform**

  In order to scale and position the *Window's* content as described in Multiple resolutions, each Window contains a *window transform*. It is for example responsible for the black bars at the *Window's* sides so that the *Viewport* is displayed with a fixed aspect ratio.

- **Window position**

  Every *Window* also has a position to describe its position within its embedder. The embedder can be another *Viewport* or the OS Window Manager.

- **SubViewportContainer shrink transform**

  stretch together with stretch_shrink declare for a *SubViewportContainer* if and by what integer factor the contained *SubViewport* should be scaled in comparison to the container's size.

Scripting development

GDScript

Annotation guidelines

Create annotations for modifiers that act on the script or its code. Additionally, create annotations for behavior that is specific to the Godot engine and editor; if the primary purpose is to affect the way that the engine or editor treats or interacts with the script, implement the token as an annotation.

Do not create annotations for general programming language features.

```
# Affects how the editor treats this script.
@icon("res://path/to/class/icon.svg")

# Affects how the engine interacts with this script.
@onready var character_name = $Label

# static is a general programming language feature.
static var num_players = 2
```

For historical reasons, some existing annotations and keywords do not strictly follow these guidelines. Choosing between implementing a feature as an annotation or as a language keyword is a nuanced decision that should be made through discussion with other GDScript developers.

Custom modules in C++

Modules

Godot allows extending the engine in a modular way. New modules can be created and then enabled/disabled. This allows for adding new engine functionality at every level without modifying the core, which can be split for use and reuse in different modules.

Modules are located in the `modules/` subdirectory of the build system. By default, dozens of modules are enabled, such as GDScript (which, yes, is not part of the base engine), the Mono runtime, a regular expressions module, and others. As many new modules as desired can be created and combined. The SCons build system will take care of it transparently.

What for?

While it's recommended that most of a game be written in scripting (as it is an enormous time saver), it's perfectly possible to use C++ instead. Adding C++ modules can be useful in the following scenarios:

- Binding an external library to Godot (like PhysX, FMOD, etc).
- Optimize critical parts of a game.
- Adding new functionality to the engine and/or editor.
- Porting an existing game to Godot.
- Write a whole, new game in C++ because you can't live without C++.

Creating a new module

Before creating a module, make sure to download the source code of Godot and compile it.

To create a new module, the first step is creating a directory inside `modules/`. If you want to maintain the module separately, you can checkout a different VCS into modules and use it.

The example module will be called "summator" (`godot/modules/summator`). Inside we will create a summator class:

```
/* summator.h */

#ifndef SUMMATOR_H
#define SUMMATOR_H

#include "core/object/ref_counted.h"

class Summator : public RefCounted {
    GDCLASS(Summator, RefCounted);

    int count;

protected:
    static void _bind_methods();

public:
    void add(int p_value);
    void reset();
    int get_total() const;

    Summator();
};

#endif // SUMMATOR_H
```

And then the cpp file.

```
/* summator.cpp */

#include "summator.h"

```

```
void Summator::add(int p_value) {
    count += p_value;
}

void Summator::reset() {
    count = 0;
}

int Summator::get_total() const {
    return count;
}

void Summator::_bind_methods() {
    ClassDB::bind_method(D_METHOD("add", "value"),
&Summator::add);
    ClassDB::bind_method(D_METHOD("reset"), &Summator::reset);
    ClassDB::bind_method(D_METHOD("get_total"),
&Summator::get_total);
}

Summator::Summator() {
    count = 0;
}
```

Then, the new class needs to be registered somehow, so two more files need to be created:

```
register_types.h
register_types.cpp
```

Important

These files must be in the top-level folder of your module (next to your `SCsub` and `config.py` files) for the module to be registered properly.

These files should contain the following:

```
/* register_types.h */

#include "modules/register_module_types.h"

void initialize_summator_module(ModuleInitializationLevel
p_level);
void uninitialize_summator_module(ModuleInitializationLevel
p_level);
/* yes, the word in the middle must be the same as the module
folder name */
```

```
/* register_types.cpp */

#include "register_types.h"

#include "core/object/class_db.h"
#include "summator.h"

void initialize_summator_module(ModuleInitializationLevel
p_level) {
    if (p_level != MODULE_INITIALIZATION_LEVEL_SCENE) {
        return;
    }
    ClassDB::register_class<Summator>();
}

void uninitialize_summator_module(ModuleInitializationLevel
p_level) {
    if (p_level != MODULE_INITIALIZATION_LEVEL_SCENE) {
        return;
    }
   // Nothing to do here in this example.
}
```

Next, we need to create an `SCsub` file so the build system compiles this module:

```
# SCsub

Import('env')
```

```
env.add_source_files(env.modules_sources, "*.cpp") # Add all
cpp files to the build

```

With multiple sources, you can also add each file individually to a Python string list:

```
src_list = ["summator.cpp", "other.cpp", "etc.cpp"]
env.add_source_files(env.modules_sources, src_list)
```

This allows for powerful possibilities using Python to construct the file list using loops and logic statements. Look at some modules that ship with Godot by default for examples.

To add include directories for the compiler to look at you can append it to the environment's paths:

```
env.Append(CPPPATH=["mylib/include"]) # this is a relative path
env.Append(CPPPATH=["#myotherlib/include"]) # this is an
'absolute' path
```

If you want to add custom compiler flags when building your module, you need to clone `env` first, so it won't add those flags to whole Godot build (which can cause errors). Example `SCsub` with custom flags:

```
# SCsub

Import('env')

module_env = env.Clone()
module_env.add_source_files(env.modules_sources, "*.cpp")
# Append CCFLAGS flags for both C and C++ code.
module_env.Append(CCFLAGS=['-O2'])
# If you need to, you can:
# - Append CFLAGS for C code only.
# - Append CXXFLAGS for C++ code only.
```

And finally, the configuration file for the module, this is a Python script that must be named `config.py`:

```
# config.py

def can_build(env, platform):
    return True

def configure(env):
    pass
```

The module is asked if it's OK to build for the specific platform (in this case, `True` means it will build for every platform).

And that's it. Hope it was not too complex! Your module should look like this:

```
godot/modules/summator/config.py
godot/modules/summator/summator.h
godot/modules/summator/summator.cpp
godot/modules/summator/register_types.h
godot/modules/summator/register_types.cpp
godot/modules/summator/SCsub
```

You can then zip it and share the module with everyone else. When building for every platform (instructions in the previous sections), your module will be included.

Note

There is a parameter limit of 5 in C++ modules for things such as subclasses. This can be raised to 13 by including the header file `core/method_bind_ext.gen.inc`.

Using the module

You can now use your newly created module from any script:

GDScript

```
var s = Summator.new()
s.add(10)
s.add(20)
s.add(30)
print(s.get_total())
s.reset()
```

The output will be `60`.

See also

The previous Summator example is great for small, custom modules, but what if you want to use a larger, external library? Refer to Binding to external libraries for details about binding to external libraries.

Warning

If your module is meant to be accessed from the running project (not just from the editor), you must also recompile every export template you plan to use, then specify the path to the custom template in each export preset. Otherwise, you'll get errors when running the project as the module isn't compiled in the export template. See the Compiling pages for more information.

Compiling a module externally

Compiling a module involves moving the module's sources directly under the engine's `modules/` directory. While this is the most straightforward way to compile a module, there are a couple of reasons as to why this might not be a practical thing to do:

1. Having to manually copy modules sources every time you want to compile the engine with or without the module, or taking

additional steps needed to manually disable a module during compilation with a build option similar to `module_summator_enabled=no`. Creating symbolic links may also be a solution, but you may additionally need to overcome OS restrictions like needing the symbolic link privilege if doing this via script.
2. Depending on whether you have to work with the engine's source code, the module files added directly to `modules/` changes the working tree to the point where using a VCS (like `git`) proves to be cumbersome as you need to make sure that only the engine-related code is committed by filtering changes.

So if you feel like the independent structure of custom modules is needed, lets take our "summator" module and move it to the engine's parent directory:

```
mkdir ../modules
mv modules/summator ../modules
```

Compile the engine with our module by providing `custom_modules` build option which accepts a comma-separated list of directory paths containing custom C++ modules, similar to the following:

```
scons custom_modules=../modules
```

The build system shall detect all modules under the `../modules` directory and compile them accordingly, including our "summator" module.

Warning

Any path passed to `custom_modules` will be converted to an absolute path internally as a way to distinguish between custom and built-in modules. It means that things like generating module documentation may rely on a specific path structure on your machine.

See also

Introduction to the buildsystem - Custom modules build option.

Customizing module types initialization

Modules can interact with other built-in engine classes during runtime and even affect the way core types are initialized. So far, we've been using `register_summator_types` as a way to bring in module classes to be available within the engine.

A crude order of the engine setup can be summarized as a list of the following type registration methods:

```
preregister_module_types();
preregister_server_types();
register_core_singletons();
register_server_types();
register_scene_types();
EditorNode::register_editor_types();
register_platform_apis();
register_module_types();
initialize_physics();
initialize_navigation_server();
register_server_singletons();
register_driver_types();
ScriptServer::init_languages();
```

Our `Summator` class is initialized during the `register_module_types()` call. Imagine that we need to satisfy some common module run-time dependency (like singletons), or allow us to override existing engine method callbacks before they can be assigned by the engine itself. In that case, we want to ensure that our module classes are registered *before* any other built-in type.

This is where we can define an optional `preregister_summator_types()` method which will be called before anything else during the `preregister_module_types()` engine setup stage.

We now need to add this method to `register_types` header and source files:

```
/* register_types.h */

#define MODULE_SUMMATOR_HAS_PREREGISTER
void preregister_summator_types();

void register_summator_types();
void unregister_summator_types();
```

Note

Unlike other register methods, we have to explicitly define `MODULE_SUMMATOR_HAS_PREREGISTER` to let the build system know what relevant method calls to include at compile time. The module's name has to be converted to uppercase as well.

```
/* register_types.cpp */

#include "register_types.h"

#include "core/object/class_db.h"
#include "summator.h"

void preregister_summator_types() {
    // Called before any other core types are registered.
    // Nothing to do here in this example.
}

void register_summator_types() {
    ClassDB::register_class<Summator>();
}
```

```
void unregister_summator_types() {
   // Nothing to do here in this example.
}
```

Improving the build system for development

Warning

This shared library support is not designed to support distributing a module to other users without recompiling the engine. For that purpose, use a GDExtension instead.

So far, we defined a clean SCsub that allows us to add the sources of our new module as part of the Godot binary.

This static approach is fine when we want to build a release version of our game, given we want all the modules in a single binary.

However, the trade-off is that every single change requires a full recompilation of the game. Even though SCons is able to detect and recompile only the file that was changed, finding such files and eventually linking the final binary takes a long time.

The solution to avoid such a cost is to build our own module as a shared library that will be dynamically loaded when starting our game's binary.

```
# SCsub

Import('env')

sources = [
    "register_types.cpp",
    "summator.cpp"
]

# First, create a custom env for the shared library.
module_env = env.Clone()

# Position-independent code is required for a shared library.
module_env.Append(CCFLAGS=['-fPIC'])

# Don't inject Godot's dependencies into our shared library.
module_env['LIBS'] = []

# Define the shared library. By default, it would be built in
the module's
# folder, however it's better to output it into `bin` next to
the
# Godot binary.
shared_lib = module_env.SharedLibrary(target='#bin/summator',
source=sources)

# Finally, notify the main build environment it now has our
shared library
# as a new dependency.

# LIBPATH and LIBS need to be set on the real "env" (not the
clone)
# to link the specified libraries to the Godot executable.

env.Append(LIBPATH=['#bin'])

# SCons wants the name of the library with it custom suffixes
# (e.g. ".linuxbsd.tools.64") but without the final ".so".
shared_lib_shim = shared_lib[0].name.rsplit('.', 1)[0]
env.Append(LIBS=[shared_lib_shim])
```

Once compiled, we should end up with a `bin` directory containing both the `godot*` binary and our `libsummator*.so`. However given the .so is not in a standard directory (like `/usr/lib`), we have to help our binary find it during runtime with the `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH="$PWD/bin/"
./bin/godot*
```

Note

You have to `export` the environment variable. Otherwise, you won't be able to run your project from the editor.

On top of that, it would be nice to be able to select whether to compile our module as shared library (for development) or as a part of the Godot binary (for release). To do that we can define a custom flag to be passed to SCons using the `ARGUMENT` command:

```
# SCsub

Import('env')

sources = [
    "register_types.cpp",
    "summator.cpp"
]

module_env = env.Clone()
module_env.Append(CCFLAGS=['-O2'])

if ARGUMENTS.get('summator_shared', 'no') == 'yes':
    # Shared lib compilation
    module_env.Append(CCFLAGS=['-fPIC'])
    module_env['LIBS'] = []
    shared_lib =
module_env.SharedLibrary(target='#bin/summator',
source=sources)
    shared_lib_shim = shared_lib[0].name.rsplit('.', 1)[0]
    env.Append(LIBS=[shared_lib_shim])
    env.Append(LIBPATH=['#bin'])
else:
    # Static compilation
    module_env.add_source_files(env.modules_sources, sources)
```

Now by default `scons` command will build our module as part of Godot's binary and as a shared library when passing `summator_shared=yes`.

Finally, you can even speed up the build further by explicitly specifying your shared module as target in the SCons command:

```
scons summator_shared=yes platform=linuxbsd
bin/libsummator.linuxbsd.tools.64.so
```

Writing custom documentation

Writing documentation may seem like a boring task, but it is highly recommended to document your newly created module to make it easier for users to benefit from it. Not to mention that the code you've written one year ago may become indistinguishable from the code that was written by someone else, so be kind to your future self!

There are several steps in order to setup custom docs for the module:

1. Make a new directory in the root of the module. The directory name can be anything, but we'll be using the `doc_classes` name throughout this section.

2. Now, we need to edit `config.py`, add the following snippet:

   ```
   def get_doc_path():
       return "doc_classes"

   def get_doc_classes():
       return [
           "Summator",
       ]
   ```

The `get_doc_path()` function is used by the build system to determine the location of the docs. In this case, they will be located in the `modules/summator/doc_classes` directory. If you don't define this, the doc path for your module will fall back to the main `doc/classes` directory.

The `get_doc_classes()` method is necessary for the build system to know which registered classes belong to the module. You need to list all of your classes here. The classes that you don't list will end up in the main `doc/classes` directory.

Tip

You can use Git to check if you have missed some of your classes by checking the untracked files with `git status`. For example:

```
git status
```

Example output:

```
Untracked files:
    (use "git add <file>..." to include in what will be
committed)

    doc/classes/MyClass2D.xml
    doc/classes/MyClass4D.xml
    doc/classes/MyClass5D.xml
    doc/classes/MyClass6D.xml
    ...
```

3. Now we can generate the documentation:

We can do this via running Godot's doctool i.e. `godot --doctool <path>`, which will dump the engine API reference to the given `<path>` in XML format.

In our case we'll point it to the root of the cloned repository. You can point it to an another folder, and just copy over the files that you need.

Run command:

```
bin/<godot_binary> --doctool .
```

Now if you go to the `godot/modules/summator/doc_classes` folder, you will see that it contains a `Summator.xml` file, or any other classes, that you referenced in your `get_doc_classes` function.

Edit the file(s) following Class reference primer and recompile the engine.

Once the compilation process is finished, the docs will become accessible within the engine's built-in documentation system.

In order to keep documentation up-to-date, all you'll have to do is simply modify one of the XML files and recompile the engine from now on.

If you change your module's API, you can also re-extract the docs, they will contain the things that you previously added. Of course if you point it to your godot folder, make sure you don't lose work by extracting older docs from an older engine build on top of the newer ones.

Note that if you don't have write access rights to your supplied `<path>`, you might encounter an error similar to the following:

```
ERROR: Can't write doc file: docs/doc/classes/@GDScript.xml
   At: editor/doc/doc_data.cpp:956
```

Writing custom unit tests

It's possible to write self-contained unit tests as part of a C++ module. If you are not familiar with the unit testing process in Godot yet, please refer to Unit testing.

The procedure is the following:

1. Create a new directory named `tests/` under your module's root:

```
cd modules/summator
mkdir tests
cd tests
```

2. Create a new test suite: `test_summator.h`. The header must be prefixed with `test_` so that the build system can collect it and include it as part of the `tests/test_main.cpp` where the tests are run.
3. Write some test cases. Here's an example:

```
// test_summator.h
#ifndef TEST_SUMMATOR_H
#define TEST_SUMMATOR_H

#include "tests/test_macros.h"

#include "modules/summator/summator.h"

namespace TestSummator {

TEST_CASE("[Modules][Summator] Adding numbers") {
    Ref<Summator> s = memnew(Summator);
    CHECK(s->get_total() == 0);

    s->add(10);
    CHECK(s->get_total() == 10);

    s->add(20);
    CHECK(s->get_total() == 30);

    s->add(30);
    CHECK(s->get_total() == 60);
    s->reset();
    CHECK(s->get_total() == 0);
}

} // namespace TestSummator

#endif // TEST_SUMMATOR_H
```

4. Compile the engine with `scons tests=yes`, and run the tests with the following command:

```
./bin/<godot_binary> --test --source-file="*test_summator*" --success
```

You should see the passing assertions now.

Adding custom editor icons

Similarly to how you can write self-contained documentation within a module, you can also create your own custom icons for classes to appear in the editor.

For the actual process of creating editor icons to be integrated within the engine, please refer to Editor icons first.

Once you've created your icon(s), proceed with the following steps:

1. Make a new directory in the root of the module named `icons`. This is the default path for the engine to look for module's editor icons.
2. Move your newly created `svg` icons (optimized or not) into that folder.
3. Recompile the engine and run the editor. Now the icon(s) will appear in editor's interface where appropriate.

If you'd like to store your icons somewhere else within your module, add the following code snippet to `config.py` to override the default path:

```
def get_icons_path():
    return "path/to/icons"
```

Summing up

Remember to:

- Use `GDCLASS` macro for inheritance, so Godot can wrap it.
- Use `_bind_methods` to bind your functions to scripting, and to allow them to work as callbacks for signals.
- **Avoid multiple inheritance for classes exposed to Godot**, as `GDCLASS` doesn't support this. You can still use multiple inheritance in your own classes as long as they're not exposed to Godot's scripting API.

But this is not all, depending what you do, you will be greeted with some (hopefully positive) surprises.

- If you inherit from Node (or any derived node type, such as Sprite2D), your new class will appear in the editor, in the inheritance tree in the "Add Node" dialog.
- If you inherit from Resource, it will appear in the resource list, and all the exposed properties can be serialized when saved/loaded.
- By this same logic, you can extend the Editor and almost any area of the engine.

Binding to external libraries

Modules

The Summator example in Custom modules in C++ is great for small, custom modules, but what if you want to use a larger, external library? Let's look at an example using Festival [https://www.cstr.ed.ac.uk/projects/festival/], a speech synthesis (text-to-speech) library written in C++.

To bind to an external library, set up a module directory similar to the Summator example:

```
godot/modules/tts/
```

Next, you will create a header file with a TTS class:

```
/* tts.h */

#ifndef GODOT_TTS_H
#define GODOT_TTS_H

#include "core/object/ref_counted.h"

class TTS : public RefCounted {
    GDCLASS(TTS, RefCounted);

protected:
    static void _bind_methods();

public:
    bool say_text(String p_txt);

    TTS();
};
```

```
#endif // GODOT_TTS_H
```

And then you'll add the cpp file.

```
/* tts.cpp */

#include "tts.h"

#include <festival.h>

bool TTS::say_text(String p_txt) {

    //convert Godot String to Godot CharString to C string
    return festival_say_text(p_txt.ascii().get_data());
}

void TTS::_bind_methods() {

    ClassDB::bind_method(D_METHOD("say_text", "txt"),
&TTS::say_text);
}

TTS::TTS() {
    festival_initialize(true, 210000); //not the best way to do
it as this should only ever be called once.
}
```

Just as before, the new class needs to be registered somehow, so two more files need to be created:

```
register_types.h
register_types.cpp
```

Important

These files must be in the top-level folder of your module (next to your `SCsub` and `config.py` files) for the module to be registered properly.

These files should contain the following:

```
/* register_types.h */

void initialize_tts_module(ModuleInitializationLevel p_level);
void uninitialize_tts_module(ModuleInitializationLevel
p_level);
/* yes, the word in the middle must be the same as the module
folder name */
```

```
/* register_types.cpp */

#include "register_types.h"

#include "core/object/class_db.h"
#include "tts.h"

void initialize_tts_module(ModuleInitializationLevel p_level) {
    if (p_level != MODULE_INITIALIZATION_LEVEL_SCENE) {
        return;
    }
    ClassDB::register_class<TTS>();
}

void uninitialize_tts_module(ModuleInitializationLevel p_level)
{
    // Nothing to do here in this example.
}
```

Next, you need to create an `SCsub` file so the build system compiles this module:

```
# SCsub

Import('env')

env_tts = env.Clone()
env_tts.add_source_files(env.modules_sources, "*.cpp") # Add
all cpp files to the build
```

You'll need to install the external library on your machine to get the .a library files. See the library's official documentation for specific instructions on how to do this for your operation system. We've included the installation commands for Linux below, for reference.

```
sudo apt-get install festival festival-dev <-- Installs
festival and speech_tools libraries
apt-cache search festvox-* <-- Displays list of voice packages
sudo apt-get install festvox-don festvox-rablpc16k festvox-
kallpc16k festvox-kdlpc16k <-- Installs voices
```

Important

The voices that Festival uses (and any other potential external/3rd-party resource) all have varying licenses and terms of use; some (if not most) of them may be be problematic with Godot, even if the Festival Library itself is MIT License compatible. Please be sure to check the licenses and terms of use.

The external library will also need to be installed inside your module to make the source files accessible to the compiler, while also keeping the module code self-contained. The festival and speech_tools libraries can be installed from the modules/tts/ directory via git using the following commands:

```
git clone https://github.com/festvox/festival
git clone https://github.com/festvox/speech_tools
```

If you don't want the external repository source files committed to your repository, you can link to them instead by adding them as submodules (from within the modules/tts/ directory), as seen below:

```
git submodule add https://github.com/festvox/festival
git submodule add https://github.com/festvox/speech_tools
```

Important

Please note that Git submodules are not used in the Godot repository. If you are developing a module to be merged into the main Godot repository, you should not use submodules. If your module doesn't get merged in, you can always try to implement the external library as a GDExtension.

To add include directories for the compiler to look at you can append it to the environment's paths:

```
# These paths are relative to /modules/tts/
env_tts.Append(CPPPATH=["speech_tools/include",
"festival/src/include"])

# LIBPATH and LIBS need to be set on the real "env" (not the
clone)
# to link the specified libraries to the Godot executable.

# This is a path relative to /modules/tts/ where your .a
libraries reside.
# If you are compiling the module externally (not in the godot
source tree),
# these will need to be full paths.
env.Append(LIBPATH=['libpath'])

# Check with the documentation of the external library to see
which library
# files should be included/linked.
env.Append(LIBS=['Festival', 'estools', 'estbase',
'eststring'])
```

If you want to add custom compiler flags when building your module, you need to clone *env* first, so it won't add those flags to whole Godot build (which can cause errors). Example *SCsub* with custom flags:

```
# SCsub

Import('env')

env_tts = env.Clone()
env_tts.add_source_files(env.modules_sources, "*.cpp")
# Append CCFLAGS flags for both C and C++ code.
```

```
env_tts.Append(CCFLAGS=['-O2'])
# If you need to, you can:
# - Append CFLAGS for C code only.
# - Append CXXFLAGS for C++ code only.
```

The final module should look like this:

```
godot/modules/tts/festival/
godot/modules/tts/libpath/libestbase.a
godot/modules/tts/libpath/libestools.a
godot/modules/tts/libpath/libeststring.a
godot/modules/tts/libpath/libFestival.a
godot/modules/tts/speech_tools/
godot/modules/tts/config.py
godot/modules/tts/tts.h
godot/modules/tts/tts.cpp
godot/modules/tts/register_types.h
godot/modules/tts/register_types.cpp
godot/modules/tts/SCsub
```

Using the module

You can now use your newly created module from any script:

```
var t = TTS.new()
var script = "Hello world. This is a test!"
var is_spoken = t.say_text(script)
print('is_spoken: ', is_spoken)
```

And the output will be `is_spoken: True` if the text is spoken.

Custom Godot servers

Introduction

Godot implements multi-threading as servers. Servers are daemons which manage data, process it, and push the result. Servers implement the mediator pattern which interprets resource ID and process data for the engine and other modules. In addition, the server claims ownership for its RID allocations.

This guide assumes the reader knows how to create C++ modules and Godot data types. If not, refer to Custom modules in C++.

References

- Why does Godot use servers and RIDs? [https://godotengine.org/article/why-does-godot-use-servers-and-rids]
- Singleton pattern [https://en.wikipedia.org/wiki/Singleton_pattern]
- Mediator pattern [https://en.wikipedia.org/wiki/Mediator_pattern]

What for?

- Adding artificial intelligence.
- Adding custom asynchronous threads.
- Adding support for a new input device.
- Adding writing threads.
- Adding a custom VoIP protocol.
- And more...

Creating a Godot server

At minimum, a server must have a static instance, a sleep timer, a thread loop, an initialization state and a cleanup procedure.

```
#ifndef HILBERT_HOTEL_H
#define HILBERT_HOTEL_H

#include "core/object/object.h"
#include "core/os/thread.h"
#include "core/os/mutex.h"
#include "core/templates/list.h"
#include "core/templates/rid.h"
#include "core/templates/set.h"
#include "core/variant/variant.h"

class HilbertHotel : public Object {
    GDCLASS(HilbertHotel, Object);

    static HilbertHotel *singleton;
    static void thread_func(void *p_udata);

private:
    bool thread_exited;
    mutable bool exit_thread;
    Thread *thread;
    Mutex *mutex;

public:
    static HilbertHotel *get_singleton();
    Error init();
    void lock();
    void unlock();
    void finish();

protected:
    static void _bind_methods();

private:
    uint64_t counter;
    RID_Owner<InfiniteBus> bus_owner;
    //
https://github.com/godotengine/godot/blob/master/core/templates
/rid.h
```

```
    Set<RID> buses;
    void _emit_occupy_room(uint64_t room, RID rid);

public:
    RID create_bus();
    Variant get_bus_info(RID id);
    bool empty();
    bool delete_bus(RID id);
    void clear();
    void register_rooms();
    HilbertHotel();
};

#endif
```

```
#include "hilbert_hotel.h"

#include "core/variant/dictionary.h"
#include "core/os/os.h"

#include "prime_225.h"

void HilbertHotel::thread_func(void *p_udata) {

    HilbertHotel *ac = (HilbertHotel *) p_udata;
    uint64_t msdelay = 1000;

    while (!ac->exit_thread) {
        if (!ac->empty()) {
            ac->lock();
            ac->register_rooms();
            ac->unlock();
        }
        OS::get_singleton()->delay_usec(msdelay * 1000);
    }
}

Error HilbertHotel::init() {
    thread_exited = false;
    counter = 0;
    mutex = Mutex::create();
    thread = Thread::create(HilbertHotel::thread_func, this);
    return OK;
}

HilbertHotel *HilbertHotel::singleton = NULL;

HilbertHotel *HilbertHotel::get_singleton() {
    return singleton;
}

void HilbertHotel::register_rooms() {
    for (Set<RID>::Element *e = buses.front(); e; e = e-
>next()) {
        auto bus = bus_owner.getornull(e->get());

        if (bus) {
            uint64_t room = bus->next_room();
            _emit_occupy_room(room, bus->get_self());
        }
    }
}

void HilbertHotel::unlock() {
    if (!thread || !mutex) {
        return;
    }

    mutex->unlock();
}

void HilbertHotel::lock() {
    if (!thread || !mutex) {
        return;
    }

    mutex->lock();
}

void HilbertHotel::_emit_occupy_room(uint64_t room, RID rid) {
    _HilbertHotel::get_singleton()->_occupy_room(room, rid);
}

Variant HilbertHotel::get_bus_info(RID id) {
    InfiniteBus *bus = bus_owner.getornull(id);
```